PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 17/50, 7/60	A1	(11) International Publication Number: WO 00/58873 (43) International Publication Date: 5 October 2000 (05.10.00)

(21) International Application Number: PCT/US00/07795

(22) International Filing Date: 23 March 2000 (23.03.00)

(30) Priority Data:
60/126,459 26 March 1999 (26.03.99) US

(71) Applicant: OCTANE SOFTWARE, INC. [US/US]; 2929 Campus Drive #101, San Mateo, CA 94403 (US).

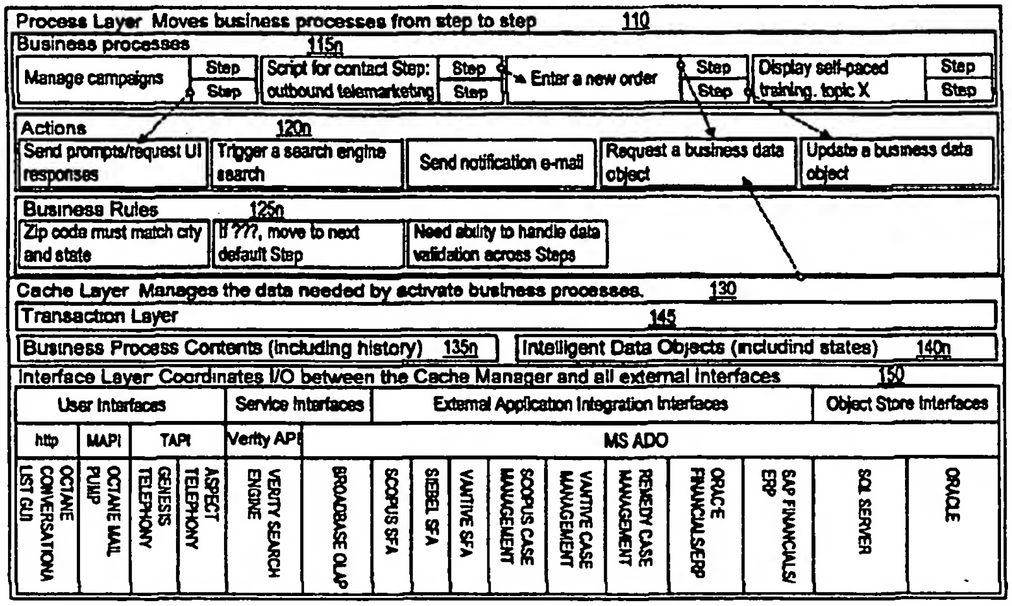
(72) Inventors: CLEMENTS, Michael, R.; 7 Madera Avenue, San Carlos, CA 94070 (US). GRYPHON, Robert, L.; 13 Tarrs Lane, Polson, MT 59860 (US).

(74) Agents: MARINO, Fabio, E. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).

(81) Designated States: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: WORKFLOW DESIGN ENGINE


Process Layer Moves business processes from step to step 110
Business processes 115n
Manage campaigns
Step
Step
Script for contact Step: outbound telemarketing
Step
Step
Enter a new order
Step
Step
Display self-paced training, topic X
Step
Step
Actions 120n
Send prompts/request UI responses
Trigger a search engine search
Send notification e-mail
Request a business data object
Update a business data object
Business Rules 125n
Zip code must match city and state
If ???, move to next default Step
Need ability to handle data validation across Steps
Cache Layer Manages the data needed by activate business processes. 130
Transaction Layer 145
Business Process Contents (including history) 135n
Intelligent Data Objects (includind states) 140n
Interface Layer Coordinates I/O between the Cache Manager and all external interfaces 150
User Interfaces
Service Interfaces
External Application Integration Interfaces
Object Store Interfaces
http
MAPI
TAPI
Verity API
MS ADO
OCTANE CONVERSATIONAL LIST GUI
OCTANE MAIL PUMP
GENESIS TELEPHONY
ASPECT TELEPHONY
VERITY SEARCH ENGINE
BROADBASE OLAP
SCOPUS SFA
SIEBEL SFA
VANTIVE SFA
SCOPUS CASE MANAGEMENT
VANTIVE CASE MANAGEMENT
REMEDY CASE MANAGEMENT
ORACLE FINANCIALS/ERP
SAP FINANCIALS/ ERP
SQL SERVER
ORACLE

(57) Abstract

A system and method for modeling business workflows comprising a process layer (110), cache layer (130), and interface layer (150). The process layer (110) moves business processes from step to step. The cache layer (130) manages the data needed by activated business processes and includes business process contexts (135n), intelligent data objects (140n), and a transaction layer (145). The interface layer (150) coordinates i/o between the cache layer (130) and external interfaces.

WORKFLOW DESIGN ENGINE

BACKGROUND OF THE INVENTION

Field of the Invention

ıne present invention relates generally to computer systems and, more specifically, to computer systems used to model workflows.

Related Art

Computer application programs have traditionally been used to model business workflows. Conventional workflow modeling systems, however, integrate business workflow modeling with the graphical user interface used to communicate with the users of the system and the data models used to store the information used as part of the workflow model. As a result, these systems are far too complex for end users to create and maintain for all but the most trivial applications and require coding changes to modify the behavior of these systems. The adoption of object-oriented programming techniques has simplified the task of modifying business workflows. Even systems developed using object-oriented programming techniques, however, still require coding changes to perform all, but the most trivial, modifications to the workflows.

This is due to the fact that, even though object-oriented programming techniques provide a tool for abstracting implementation-level details from system design, they are still part of software development tools intended to assist programmers and not end users. In addition, workflow modeling systems tend to be modeled either around the data that is used to represent the workflow, that is typically stored in some for of permanent storage (e.g., a database) or around the graphical user interface used to communicate with the users.

As a result, changes to the business workflow often require changes also to the underlying access and retrieval of the information stored in the database or to the graphical user interface (GUI) of the system.

SUMMARY OF THE INVENTION

The system and method of the present invention allow for easy and efficient modeling of business workflows by separating the workflow modeling process from the details of both the user interface and the storage of data onto the system. As a result, business analysts can concentrate on modeling the business workflow using, for example, a workflow modeling language, independently from the user interface and data storage considerations that affect prior art system. As a result, the process of building and maintaining business workflow modeling systems is greatly simplified.

This is accomplished by separating the workflow model from the user interface and data storage components of the system and using a synchronizer to coordinate the operation of the system. Using the system architecture of the present invention, business workflows can be modeled independently of the graphical user interface that is used to communicate with the users of the system. Similarly, the way data is stored in a database can be modified without necessarily requiring changes to the business workflow models.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 shows a conceptual view of a computer system architecture, in accordance with an embodiment of the invention.

Fig. 2 shows the components of a Workflow.

Fig. 3 shows a tree of related contexts in the lineage associated with a user session while the transaction server is running.

Fig. 4 illustrates how steps are added to the Stepstack either by a workflow or by other steps.

Fig. 5 shows MAPI objects represented in the CDO object model.

Fig. 6 is a block diagram of the E-mail system object collaboration.

Fig. 7 is a sequential diagram illustrating the operation of the Email Peripheral model.

Fig. 8 is a block diagram illustrating a three-level deep chain of bags.

Fig. 9 is a block diagram of the transaction server's security architecture.

Fig. 10 is a block diagram illustrating the physical architecture of a typical transaction server system that supports email processing.

Fig. 11 is a block diagram of a "one-way trust" security configuration.

Fig. 12 is a flow diagram of the possible outcomes of an operation, in accordance to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

A computer system and method of operation thereof are described, in accordance with the principles of the present invention. The computer system includes six major components:

Business Workflows (BWs), which drive the application (and frequently the user) by using Business Rules (BRs) to coordinate and sequence Operations.

Business Information (BIOs), objects that store the data used by the application. When a BIO can be saved in a storage repository it is called Persistent Business Information (PBIO). Customers, orders and addresses are all examples of PBIOs.

Mappers, which move data between persistent storage and BIOs, performing any necessary data transformations. Mappers use DBWires, which talk to databases, and they use TransMaps which are data organization trees.

Interactors, which render applications (consisting of Workflows and Business Information objects) on various delivery channels such as Web, Email and Telephony.

Synchronizer, which controls and sequences the operations of the entire system. The Synchronizer is an event machine that enables the various independent objects in the system to collaborate anonymously.

Peripheral, which orchestrates the cooperation of the transaction server with external devices and agents, such as CTI and Email.

This document lays out only the high level design requirements of the architecture of a computer system in accordance with an embodiment of the present invention. The architecture design and all other detailed aspects of this software are modeled in UML and other appropriate design tools.

The system architecture described herein is a "framework" design. The system provides a framework in which one quickly can build robust and reusable business applications without having a lot of technical knowledge.

The six most fundamental entities in the system are Business Workflows, Business Information Objects, Mappers, Interactors, Synchronizer, and Peripherals.

The first section provides an overview of the architectural design and the following sections will describe in detail these fundamental entities.

Conceptual Architecture

Fig. 1 shows a conceptual view of a system architecture in accordance with an embodiment of the invention. Object categories are illustrated in an overview of their structural relationships and collaborations.

The system architecture of Fig. 1 includes three layers: process layer 110, cache layer 130 and interface layer 150.

Process layer 110, in turn, includes business processes 115n (where n = A, B, C, etc.), actions 120n and business rules

125n. Process layer 110 is responsible for modeling and controlling business workflows.

Cache layer 130 includes business process contexts 135n, intelligent data objects 140n and a transaction layer 145. Cache layer 130 stores information received from interface layer 150 and manipulated by process layer 110 during execution of the business workflows. When processing is completed, the data stored in cache layer 130 can be stored back into external storage facilities via interface layer 150.

Finally, interface layer 150 coordinates input and output of data between cache layer 130 and any external interfaces to the system.

<u>Abstraction Layers</u>

The architecture consists of six fundamental abstraction layers or models:

- Relational/Persistence Data
- Representational/Business Information
- Behavioral
- Interface/Contact Channel
- Synchronizer
- Peripheral
- Relational/Persistence Model

The framework data model de-couples runtime dynamic data models (representational data models) from their persistent structure (relational models). The Relational/Persistence model includes the objects used to communicate with persistent storage (RDBMS), and those used to perform the bi-directional mappings between representational and relational data models.

This enables Workflows, Steps and Operations to talk only to representational data objects, which greatly simplifies the architecture by completely decoupling applications from persistent data structures.

Representational Data Model

This model consists primarily of Business Information Objects (henceforth, BIOs). These are packaged, active, dynamic "views" of business data. Workflows and other behavioral objects can use BIOs without caring whether they are persistent or what are their persistence mapping details.

Behavioral Model

A Workflow orchestrates Steps, Operations and Rules. The Workflow is an object that provides the context under which Steps, Operations and Rules run, and tracks the history of what they have done. Fig. 2 shows the component parts of a Workflow.

In Fig. 2, business process 115A starts executing within a context 135A by interacting with plug-ins 210. As a result of this interaction, information stored in context 135A is either modified or new information is added to the context (i.e., the context grows).

Step 250A then triggers action 120A. As a result, prompts and requests for responses are sent to Conversationalist GUI 220. The result of this interaction is evaluated according to an enabled business rule 125A and, if the business rule returns a success code, context 135A is updated to reflect the results of the interaction.

Similarly, step 250B triggers action 120B to initiate a interaction with Mail pump 230. As a result of this interaction, context 135A may again be modified. Step 250C then triggers action 120C, whose results are verified by business rule 125C. When business process 115A reaches terminus 260, control is transferred back from business process 115A to another business process.

Contact Channel Model

The architecture enables a business analyst to define canonical applications. That is, an application consists of a behavioral model (what it does) with the business data objects it manipulates (what it is). The way the application is presented or rendered can be mere ornamentation, not an intrinsic part of the application.

Some examples of contact channels are Web, Email, CTI and ATM machines. Because the Workflow's behavior and data models are already defined, Contact Channels merely provide a means to communicate this information with the application user. Indeed, many Workflows run without ever knowing or caring about which contact channel is being used. Indeed, different instances of the same Workflow might be rendered simultaneously to different users on different contact channels.

Server/Synchronizer Model

The Synchronizer drives workflow and orchestrates system activities. It ensures that actions transpire in a flexible, yet orderly sequence based on Workflows designed by the business analyst, choices made by the user, error conditions encountered, and so forth. Workflows can collaborate anonymously through the Synchronizer.

Peripheral Model

Peripherals are external devices or agents that cooperate with the transaction server. Two examples are Email and CTI. Some peripherals can be interactive and thus also may act as Contact Channels. The peripheral model orchestrates the cooperation between transaction server and these external devices by providing basic functionality such as sending email from a Workflow or automatic filtering or responses to messages. The peripheral model may forward some events to the Synchronizer for processing by Workflows and business Rules.

Object Classification

All objects principally have a Class name and an Instance name. The class name is often referred to as an object's type, though it also can be regarded as the set of interfaces the object supports. The instance name is the name of a particular instance of the object. For example, three different objects may have a class name of integer but have instance names of x, y and z. Though all object oriented software systems share this concept, in the Architecture it is made explicit.

All objects in the Architecture implement the ICoreObject interface. This means they are derived from class ICoreObject and aggregate an instance of class CoreObject. Thus they inherit a common interface - and provide default behavior - that can be used throughout the Foundation. This permits Studio and other tools to change in a controlled and coherent manner when we introduce new objects or if we rename an object. Additionally, it confines all such changes to the pertinent objects (ICoreObject and CoreObject), protecting the architecture from such changes (no fragile base classes). The ICoreObject interface consists primarily of the explicit Class and Instance names described above, though in the future this interface may grow.

Class Types, Names, Instances and Meta-data

Objects in the Framework can be described in several domains: a class, a class type, a class name and an instance name. Anyone discussing the framework will tend (for convenience) to use identifying terms from different domains. The purpose of this section is to clarify the potential confusion this can cause.

Classes

A class refers to a specific technical category of objects described in source code, an actual C++ class. For example, CBD is a class. Classes are only meaningful to core developers who build the framework.

Classes are the core of the ATL/COM objects that comprise the system, are designed and coded by the core development team, and are hard coded.

Class Type

A class type is a word or phrase that describes a class. For example, BIO is the class type for Business Information, and Step is the class type for a Step in a Workflow. A class type is a way of referring to a category of objects in the framework.

Class Types are designed and coded by the core development team in accordance with requirements set by the application development and business analyst teams, and they are hard coded into the framework.

Class Name

A class name is a word or phrase that describes a class used within a specific context. A class name describes a specific category of a class type. For example, Customer is the class name for a Business Information Object used to store a customer. Where class type is general, class name is specific. However, class name still identifies only a category of objects, not a specific, unique object.

Each Class Name describes a specific variation of a Class Type. Class Names are defined by business analysts and defined in meta-data; they are not hard coded. They can be designed, created, destroyed and modified in meta-data without modifying or rebuilding the framework.

Instance Name

An instance name is a word or phrase that describes a specific instance of an object. For example, either JoeBloe or -73 may be the instance name for one particular Customer object.

Instance Names are defined arbitrarily by business analysts through Business Rules and schema requirements.

Example

As a summary example, consider a particular Business Information Object in the Transaction Server that is storing the customer whose name is Joe Bloe. How would we describe this object?

The class is CBD, the C++ class that implements the BIO class type.

The class type is PBIO, for Persistent Business Information Object.

The class name is PBIO.Customer, since it is storing a customer.

The instance name is JoeBloe, since it is storing that customer.

Relational/Persistence Model

The persistence model is the abstraction layer that provides services for access to persistent storage. It contains two parts. One is the Database Wire (DBWire), which is used to connect to a database and submit SQL statements. Another is the Persistent Business Information Map (PBDMap) that contains information about where the business data lives in a persistent storage; this object uses a DBWire to store data to or retrieve data from the persistent store. Note that PBDMap is not the only component that uses the DBWire.

Any component in the Architecture that communicates with persistent storage can use the DBWire. However, to keep the abstraction models clean, to improve the maintainability of the system, and to avoid fragile object dependencies, the Architecture minimizes the number of objects that communicate with persistent stores. Indeed, the only objects that do this are the PBDMap objects. All other objects talk to the data model. Thus, the data model presents a layer of abstraction or firewall which isolates all other components from the details of persistent storage.

Database Wire

DBWires provide an abstract interface for objects to connect to a RDBMS and store data to or retrieve data from the RDBMS. Typically the DBWire lives in the transaction server machine, the RDBMS lives on the database server, and the client components using the DBWire live in the transaction server machine (in-proc with the DBWire). The current implementation of the DBWire object deploys two of the mainstream RDBMS:

Microsoft SQL Server and Oracle. To provide a dynamic, flexible and scalable system, the DBWire implementation provides a data access environment that supports multiple client connections to the RDBMS and supports heterogeneous transaction management features.

The DBWire

DBWires provide an abstract interface for objects to connect to a RDBMS and store data to or retrieve data from the RDBMS. For example, when a BIO is initiated, it uses the BDFactory to build its property list. The factory then uses a DBWire to read meta-data from the database to find the property list for the specific BIO class name and return this information back to the BIO.

All DBWire classes share the invariant methods of the IDBWire interface so that the clients of the DBWires can use them generically. The implementation of a DBWire class encapsulates the particular category of RDBMS and the interfaces to access them.

A reasonable way to implement a DBWire class is to use the most generic industrial standard interface for a category of of RDBMS. For example, the primary implementation for architecture is ADOWire, which is based on the Microsoft ADO interfaces and can be used to access a variety of RDBMS that support ODBC or OLE DB interfaces such as Microsoft SQL Server and Oracle. Note that we will not implement an ODBCWire because the ADO interfaces are more generic than the ODBC interfaces. Since the Oracle RDBMS supports the ODBC interfaces, we are not going to implement an OracleWire neither, unless there is a need to use the native Oracle interfaces, such as for performance reasons to avoid the overhead of ODBC.

A DBWire class can also be implemented for a particular RDBMS which does not support any standardized interface and the only way to access it is though a proprietary interface. Either way, all DBWire objects expose the canonical IDBWire interface so their clients are completely decoupled from the database technology used. For example, if the current objects were completely modified internally so that, for example, they use OLE DB instead of ODBC or ADO, clients of these objects would continue working, unaware of these implementation changes.

ADOWire

ADOWire is an implementation of the IDBWire interface. It is based on Microsoft's ADO (Active Data Objects) technology, which provides universal access to any tabular data that provides an OLE DB data query processor (OLE DB driver).

The current ADOWire object uses ADO 2.0, which requires ODBC level 3 drivers for the particular database being used. Currently, this object can talk to Oracle 8.05 and SQL Server 7, and Sybase is in progress. Thus, ADOWire is a single object that talks to every database that supports.

MTS, Transaction, Multiplex and Scalability

The architecture of the present invention is designed with the location transparency in mind for object components and even for RDBMS. That means we will have object components and even different RDBMS deployed in different NT Servers or Workstations. The framework uses the Microsoft Transaction Server (MTS) to manage these transactions. Unfortunately, MTS has strict requirements and incredibly high overhead for transactional objects. Because of this, a Transact object handles transaction semantics. Instead of administering transactional objects to MTS, these objects live outside MTS and use a Transact object (which lives in MTS) and pass it around to other objects that participate in the transaction. The Transact object is used as a transaction state accumulator. In a sense, this is turning MTS inside out.

As mentioned in the previous section, the ADOWire running in the MTS environment will support multiple connections to an

RDBMS with the ODBC 3.0 Driver Manager maintaining a pool of ODBC connections.

Persistent Business Information Map

A PBIO does not know where its data resides in a RDBMS, nor does it know how to persist its data to or retrieve its data from a RDBMS. A PBDMap provides an abstract interface to map the properties of a PBIO to their locations in a RDBMS and persist its data to or retrieve its data from the RBDMS. Every PBIO uses a PBDMap. When a PBIO is asked to persist its data to or retrieve its data from a RDBMS, it delegates the request to its PBDMap.

Additional information concerning the PBDMap will be found in this document's main section entitled Representational/Business Information model.

The PBDMap object itself is stateless, which allows the Architecture to leverage significant scalability improvements described below.

Property Maps

A PBDMap contains a collection of the Database Property Maps (DBPropMap) for the PBIO's properties. A DBPropMap provides an abstract interface to map a single property to its location in a RBDMS.

PBIO Map Factory

When the PBDMap initializes itself, it delegates the task to the PBDMapFactory. The PBDMapFacoty is a singleton object that caches the property maps for the PBIOs.

The PBDMapFactory implements the PBDMapFactory interface which has only one method: init(). The PBDMap calls this method to get a bag of property maps for the PBIO. If the PBDMapFactory has the property maps in the cache, it returns the property maps to the PBDMap. Otherwise, it delegates the request to a master factory to build the property maps and caches the returned maps. The master factory in turn delegates the request to the PBDMapLoader.

PBIO Map Loader

A PBDMapLoader is the only component that knows the meta-data describing where the properties of the PBIO live in the persistent storage. It uses a DBWire (described below) access to the RDBMS and build the property maps. For the PBIO.

The PBDMapLoader implements the IMetaLoader interface which has only one method: getBag() which builds a bag of property maps for the PBIO.

Representational/Business Information Model

A BIO is a group of related data united around a single concept. Common examples of BIOs are Customers, Employees, and Orders. A BIO is a container that organizes and "cooks" or packages business data so that it can be used in a BW.

A BIO Class is a template for a particular type of BIO. The term "BIO" most often describes an instance of the BIO class (rather than the class itself), although the two meanings are frequently used interchangeably.

BIO Classes are extremely flexible. They can reuse properties, and they can be properties. All BIO Classes share the invariant common methods Init, Release, Set and Get. BIOs can encapsulate other BIOs, yet the encapsulated objects can act on their own as well.

BIOs may be persistent. A Persistent BIO (PBIO) has an extended interface that supports methods Retrieve and Persist. They implement these methods by reorganizing their information to the format of a persistent store and storing it there with a Mapper. BIOs operate within the Context, and Business Rules and Operations access them from there.

BIOs maintain arbitrary state information. Business Analysts can use Workflows to define state models for business data; the state of such a model is stored in BIOs. Business Analysts can also define state models for business process; this is described below, in the section regarding behavior.

BIOs control access to their methods via wrappers of Business Rules. This is a final line of defense against incorrect input or operations inappropriate for a given state of the BIO. Business Rules can use the state information in BIOs as input indicating whether operations should proceed.

Persistent Business Information Map (PBDMap)

A PBIO does not know where its data resides in a RDBMS, nor does it know how to persist or retrieve its data. The PBIO delegates these operations to a PBDMap. The PBDMap provides an abstract interface to map the properties of a PBIO to their locations in a RDBMS and persist its data to or retrieve its data from the RBDMS. When a PBIO is asked to persist its data to or retrieve its data from a RDBMS, it delegates the request to its PBDMap.

A PBDMap contains a collections of the Database Property Maps (DBPropMap) for the PBIO's properties. A DBPropMap provides an abstract interface to map a single property to its location in a RBDMS. It contains the Database Connection (DBConnect), Database Table (DBTable) and DataBase Column (DBCol) for the property, which store the database, table and column information for the property.

A PBDMap uses a DBWire (described below) to access to the RDBMS. The PBDMap contains the logic behind reorganizing the information in a PBIO; the DBWire is used to push or pull this information from a database.

Composition of a BIO Class

Most of the properties in a BIO Class are data fields originally derived from user input or from the database.

The properties within a particular BIO Class originate from several different sources:

Some are native to this BIO Class.

Some properties are part of a related BIO Class, but are also considered part of the current BIO. This occurs when the

BIO has complex property types, including Matrices and other BIOs.

BIO Relationships and Linking Data

Links to other business objects are the OO equivalent of 1-many or many-many relationships in relational theory. In OO this translates to the concepts of usage or containment. Because details of persistent storage do not exist in the BIO, all BIO relationships are simple 1-1 or 1-Many. Many-to-Many relationships are an implementation detail of relational databases - just a way a database can store relationships that, in the OO model, are bidirectional 1-Many. The OO data model is hierarchical where persistent storage is relational.

For example, imagine a data structure used to store cars and parts. In the persistence data model (the database), a M-M relationship would be used, as this is the most efficient way to store the data for OLTP access. But in the representational model the link table merely obfuscates the conceptual relationship between the objects. Here, bidirectional 1-M relationships would be used. When an application wants to know what parts are on a given car, all it wants to do is to go its car object and ask for a Matrix of part objects. The application does not know or care how these relationships are persisted in a database, thus forcing it to go through a M-M link table would be pleonastic.

PBDMap objects take care of this information transformation, decoupling the Representation and Persistence models so that each can be completely and fully optimized, obviating the need for the compromised data organizations required by legacy systems such as Siebel or Silknet - which work in both, but are optimized for neither.

Workflow Contexts

Contexts are available to all Steps which may be linked into the relevant BW.

Contexts may or may not be "temporarily persistent." If a BW is suspended either by the user or the Process Manager, it will be stored in the Object Store until a cleanup process weeds it out.

A Context keeps track of its Business Information. The conte.: maintains references to all the data objects (Properties of all kinds, including BIOs and Matrices) used by a Workflow. Steps can add new BIOs to the context.

A Context remembers its parent Context. When a Workflow is started by another Workflow, the latter is the parent of the former; not all Workflows have parents, since some are started by channels. When Workflow acts as a Step in a parent BW, its Context contains a reference to the Context of the parent BW (if any). Thus every Step in a BW can use the entire family history of Process Contexts. See the above section on Contexts under BWs for more information.

Workflows construct Contexts as temporary progress-tracking objects. As a BW moves through its Steps, it grows its Context. A Context maintains a record of the Steps that have been visited, the results of Business Rules consulted thus far in the BW, the Operations which were taken, the return codes of these Operations, and any other information that is pertinent to the BW.

A Context makes information available to Operations and Business Rules. Operations and Business Rules use this Context during the course of the BW. They also talk to BIOs and other data objects that "live" in the Context.

Some Contexts are "temporarily persistent" and are therefore saved in the Object Store for a certain period of time. In some cases users (or some other influence) may deliberately suspend a BW to be picked up later. Also, some types of BWs, such as those involving Internet interaction, are susceptible to users leaving the process with no notice, yet wishing to pick up where they left off in order to avoid duplicate input of data. In cases where a user might have to leave the system for a while, the application may offer a trigger mechanism (e.g. a button) that enables the user to suspend the BW. When the user reenters the BW, the User Interface Peripheral he is using must offer some way for him to automatically or manually search for his suspended Context—for example, reloading a page (with an embedded tag indicating the Context ID) in a browser, or entering a code into a telephone keypad. After a certain period, a suspended Context may expire and be deleted by a maintenance BW.

Context Preloading

Whenever a channel wants to start a new workflow it usually has information that the workflow needs. For example, all interactive channels (Web, Email, CTI, etc.) usually want to identify the person at the other end of the channel so the workflow can take appropriate action. These channels need a way to stuff information into the Context so that it will be available to the Workflow when it starts. calls this "Context Preloading".

It is essential to preload the Context without exposing to the channel details about how Contexts are created and destroyed. Thus, the channel cannot itself push data into the Context. Indeed, the Context already has interfaces designed for clients that already have data objects, so the clients can direct the Context to assume or contain these data objects. The reason channels cannot use these already-existing Context interfaces to do this, is because when the channel sends a message to start the Workflow, the Context has not yet been created!

A simpler approach is to allow the channel to create the data objects it wants to have preloaded and pass these objects to the server in the notification event that the channel was going to send anyway (to cause the workflow to start). The server picks up these data objects and inserts them into the

Context. Not only is this simpler, but it also solves the dependency inversion problem. The client channels remain dependent on the server, as they should be. Since the server never "talks" to them, it cannot become dependent on them.

One drawback to this system is that when client channels create the data objects, it is possible that the data objects will be created in an apartment different from the apartment of the Context. This would slow down system performance because it would have to marshal whenever these data objects were used by the workflow.

Events and Event Information

The purpose of a channel is to orchestrate the interaction between the transaction server and a system in the real world, such as a web browser. These real world systems are unpredictable and thus inherently event-driven. Thus it is natural that the primary means of communication between the transaction server and a channel is that the channel sends Events to the Synchronizer, and the latter returns status codes indicating how it handled the Event.

An Event describes "what happened," not "what to do about it". The former is comprised of four key pieces of information:

- Scope: where the event happened. This is usually, but not always, one of a set of predefined values, including Web Contact Channel, Messaging, etc.
- ID: what happened. This is usually, but not always, one of a set of predefined values, including Session Started (2), Session Ended (4), Session Reenter (3), etc.
- Name: Additional information describing the event.
- Info: Opaque information the event carries along with it.

The event information is completely arbitrary, though in most cases it is a pointer to the Context under which the event was thrown. Workflow Queueing on page 32 of this document contains more information describing events, event information and event processing.

How to Pre-load the Context

The framework includes a class called BDBuilder that can be used to build data objects that can be preloaded into the Context.

Context Scopes and Levels

Every workflow has a context that stores the state of the workflow's data model and its process model, and a single user session often involves many simultaneous workflows. Since these workflows must interoperate, it is useful - one might even say necessary - for each workflow to be able to access data objects that belong to the context of a different workflow.

Context Scope Rules

The system of context scopes may become quite complex at runtime, but it is governed by a few simple rules. However, before presenting the rules, we need some definitions:

Definition: Lineage

We define a lineage as the set of all workflows running under a given user session. Every session starts with a workflow, and this workflow's Context becomes the master context of the lineage. Every workflow in the lineage can always access this master context, though the reverse is not true.

Definition: Server Global Context

A Server Global Context is a singleton context created by the transaction server Synchronizer and shared by all workflows. This context is created when the transaction server is started, destroyed when the transaction server is shut down, and can be used by any workflow. Any data placed into this Context will live for the lifetime of the transaction server.

Definition: Control Local Context

Finally, a Control Local Context is a temporary context created when an Operation is fired, used for that operation and its children (if any), and destroyed when the Operation is finished. When multiple consumers all use the same operation, each consumer will have its own Control Local Context.

A Control Local Context is also created when a script is fired from a channel handler (even when no Operations are used). If the script fires any operations, those operations use the Control Local Context of the script (rather than getting their own).

Now we establish the following rules:

Every time a new workflow Y is started -hence a new context is created - this must occur from some other workflow X, and the new context Y has the context X as its parent.

The exception to this rule is the master context of the lineage, which has no parent.

Context parentage is unidirectional - parents do not know about their children.

When any workflow refers to a variable X, the transaction server first searches the control local context (if any). If X is not found here, the transaction server searches the workflow's own context. If not found, it searches the parent context. This continues iteratively until it finds X or it reaches the master context of the lineage.

If the variable X is not yet found, transaction server will search the Server Global Context.

As these rules imply, a lineage consists in general of several contexts connected in a unidirectional N-way tree with the master of the lineage at the root of the tree. Each node in the tree is a context and it can see every other context between it and the root of the tree. Two contexts, each on different branches of the tree, cannot see each other.

Context Registration and Lineage

Taking a snapshot of the lineage associated with any given user session while the transaction server is running, would reveal a tree of related contexts, as shown in Fig. 3.

On the left side of Fig. 3 we see a tree of nested contexts 310. Each points to its parent, though the parents do not know about their children. In general, given a context in the tree (such as D), it is impossible to find all other contexts in the workflow (for example, you cannot get to A from D because you cannot traverse downward). Thus, the tree cannot be enumerated or iterated. To resolve this problem we introduce the Workflow Registry 320 depicted on the right side of the diagram. The Workflow Registry stores a list of all the contexts in the lineage, ignoring their hierarchical relationships. Every lineage has exactly one Workflow Registry 320 and every context in the lineage points to the workflow registry, as shown in Fig. 3.

As the system runs and workflows are created and destroyed, the transaction server maintains the context lineage trees and workflow registries such that they are always up-to-date.

Context Scoping Extensions and Caveats

The prefixes illustrated in Table 1 below can be attached to variable names to force exceptions to the above rules. Any exception that applies to finding a variable, also applies to creating it if creation is necessary.

Table 1

Prefix	Purpose
ControlLocal.	Check only the Control Local Context. Do not check any other context.
Local.	Check the Control Local context (if any), and then the workflow's own context; do not check any parent or global contexts.
Global.	Check only the master context of the lineage. Do not check local or any other contexts.

ServerGlobal.	Check only the server global context. Do not check any other context.

<u>Context Scope Examples</u>
<u>Stories</u>

A "Story" is way of arranging BIOs to provide all information surrounding a key central idea, such as an Individual, an Organization, an Opportunity, or a Case. A Story is all the information that surrounds a central point that gives meaning to that central point over time. A story takes advantage of the hierarchical arrangement of the data model and may be cyclical.

A Story about an Individual would involve these kinds of information:

Basic information:

- Name
- Addresses
- Employee information and summary of employer, if the individual is an employee
- If the individual is a prospect, information about the campaigns and products to which he is tied
- If the individual is a customer, information about the products he has purchased, including registration information

A Story is like a snowball, picking up different kinds of information over time as it rolls along. An Individual may start out in the system as a Prospect, then become a Customer, then become a Prospect for a different Product, then we may hire her as an Employee.

Even the highest level BIOs contain only the information pertinent to a BW. But the information needed to satisfy a process-centric point of view is only a fraction of what would be needed to tell the whole story. Stories exist as extensions of their core BIOs, extending the information provided in the BIO beyond what would be needed for any given process.

For example, a BW that focuses on a Case object certainly would be interested in the particular steps taken toward resolving the case, and might be interested in the customer who opened the case, but the BW would not care about that customer's addresses. Conversely, a BW that focuses on a Customer object certainly would be interested in the customer's addresses, and might be interested in the cases that customer had pending, but it would not care about the particular steps taken toward resolving these cases.

The BW models the dynamic aspect of the application, describing how Business Process and Business Information interact. It includes data only to the extent that data is pertinent to the flow of the process. Stories provide a static aspect. They describe the complete Business Information model, and include process only to the extent that process has affected the current state of the data. Both models (static and dynamic) are critical for understanding and describing a Business Application.

A Story can be searched for by any identifying information in any of its facets, for example, "What's the story with that Individual?"

Business Information Modeling Details

Business Information presents technical design constraints that make it one of the most difficult areas to model. Some of these constraints:

All BIOs must be derived from a "master" BIO interface, so that they can be passed around and used generically.

To make it easy for a business analyst to make new types of BIOs, we cannot require the use of a compiler to generate new BIOs. Thus particular BIOs (customer, entity, account, etc.) cannot be "cooked" into hard coded objects, as this would require compiling new COM objects to make new BIOs.

The definition of BIOs, Properties, Rules and their relationships can be done in myriad ways; what is the best way?

All BIOs must be of the same actual binary class, though they may have different structure and identify themselves differently at runtime. This further implies that the structure of all BIOs must be defined in some kind of meta-data or repository. As BIOs are instantiated at run time, an object modeled as a Builder pattern constructs the particulars of the desired BIO object.

This leads to the question, "If all BIOs are of the same class, what does it mean to say that a customer BIO is different from an account BIO? In other words, what can vary from one BIO to another?"

The answer is simple but not obvious. What varies from BIO to BIO is the Property List, the Rule List and the Persistence Map. We'll describe each below. The key point is that even the most dissimilar BIOs implement the same interface using the same underlying code. Even the implementation of different BIOs is the same actual code (COM object). This is a simple object that aggregates a bag of Properties, a Rule (which may be composite), and a PBDMap object (if it is a persistent BIO). It implements the functions in the IBD interface by delegating them to the appropriate aggregated objects. For example, get is deferred to the property list and persist is delegated to the PBDMap.

BIO/PBIO initialization

When a BIO is instantiated it is generic, with no property list, no rule list, no persistence map and no identity (no classname). It is unusable. After instantiation, the BIO must be initialized. During initialization, the name of the type of BIO desired (customer, account, etc.) is passed with the init method. The init method constructs the property list, the rule list and the persistence map (if appropriate). After this, the BIO has a class identity, but does not contain any data -

except perhaps default values or lists for domain-constrained properties. Finally, in the case of a persistent BIO (a PBIO), the retrieve method can be used to retrieve data from persistent storage into the BIO.

A simple example should clarify all this. Suppose we want to create a customer PBIO and get the customer Joe Bloe.

First, we instantiate a PBIO. This gives a blank, generic data structure. Then, we call PBIO.init("customer"). This will turn the generic PBIO into a customer PBIO. Now it is a blank, uninitialized customer. Next, we call PBIO.retrieve("Joe Bloe"). This will find the customer Joe Bloe in persistent storage and load it into the PBIO.

Of course the class names and instance names will not be "customer" and "Joe Bloe" since we need names that will be unique. But they will be human readable string based identifiers. Additionally, if we don't have any unique identifiers (a customer calls in but doesn't have his account number handy), we can use a Matrix and populate to find a set of customers and pick one.

Rule List

Every BIO has a Rule that is guaranteed to be kept TRUE. The BIO will refuse any operation that would cause its Rule to be FALSE. The Rule may be a composite, may reference any property in the BIO, and has access to the Context of the current BW. Thus the Rule can see any property that every Context must have, such as operator role, security level, event list, dispatcher, variables, etc. But the rules cannot see anything outside the canonical Icontext interface, which means they cannot become dependent on extended Contexts specific to a given process.

If we assume at some time T the BIO is in a consistent state (its Rule evaluates TRUE), and its Rule can only reference the properties of the BIO itself, then the only operation that can invalidate the BIO is a set. Thus, the set method evaluates the BIO's Rule using the newly proposed property value; if the Rule returns FALSE, the set aborts and returns failure to its client.

Some rules are attached to the BIO, which means they are evaluated every time set is invoked, regardless of which property is specified. Other rules are attached to the individual properties of the BIO, which means they are evaluated only when set is invoked for that property. For maximum performance and scalability, property-level rules should be used instead of BIO-level rules wherever possible.

In the simplest case, a Property's value can be validated without reference to any other Property. Here, a Property Rule (rather than a BIO Rule) is the best method for validating this Property.

In the case where property X cannot be validated without also checking properties Y and Z, the best method for validation is a BIO Rule. Certainly, one could still attach the rule to property X - but if Y or Z changed, X may no longer be valid even though X was not changed. Since the BIO Rule evaluates every time set is called - regardless of which property is being set - it can guarantee consistency of X, Y, and Z. But one might not want to use a BIO Rule because it would be evaluated for every property set, even if it did not involve X, Y or Z. The alternative to a BIO Rule on X only, is to attach similar rules to all three properties X, Y, and Z. This is more complex, but also more scalable. The architecture remains open and flexible on this point, leaving the decision which method is best for any given situation in the hands of the business analyst or developer.

<u>Attendants</u>

Obstructing set operations that would invalidate a BIO is good first step, but we'd like to do something more useful than simply block the operation. Perhaps the data could be quickly and simply modified to make it valid. Perhaps further properties could be set, or warnings could be fired off. In general, the application designer may want to use a property set as a trigger to fire off some (lightweight) processing.

This ties in with rules in the following manner. First, the rule is fired. If the rule returns TRUE, the set proceeds with no further ado. If the rule returns FALSE, an operation is performed. If the operation succeeds, we assume the data is "fixed" and the set proceeds. If the operation fails, the set does nothing and returns failure to the client.

It turns out that this semantic is precisely that of Operation object. The only difference is that in an operation, the rule is considered an inhibitor, so the operation proceeds only if the rule returns TRUE. The boolean value is reversed.

Thus the rules of a BIO might be modeled as references to IOperation objects. The operation's inhibitor rule is used as the validity checker, and operation's action is used as the "fixer-upper" for the data. Only if both fail is the set obstructed.

Of course, since an IOperation can do "anything," this scenario would make it impossible for any set on a BIO to be a transaction. The operation might have side effects that cannot be undone - for example, sending a fax. If the set transaction is rolled back it may be impossible to "undo" the effects of the trigger. The sphere of control of the trigger must be restricted so that anything it does can become part of a transaction that lives within the BIO.

For details concerning the semantics of operations and for a complete description describing spheres of control, see the appropriate section in the Workflow section of this document.

<u>Property List</u>

The property list of a BIO contains its data. Every element in the list is a single property with a given name and elemental type. The elemental types include:

- Flag (boolean)

- String (varchar 255)
- Text (unlimited free-form string)
- Integer
- Floating point
- Currency (BCD if available)
- Time
- BIO/PBIO
- Matrix

Significantly, any element in the property list can be another BIO, a PBIO or a Matrix.

Every element in the property list can be the subject of a get or set message. Getting a property that is itself a BIO, returns a reference to the BIO. Conversely, setting a BIO property replaces it with the one provided in the set method.

Property Rules and Validation

A property's type or name cannot be changed at run time. However, properties of different types can be compared or set and the framework will attempt to perform automatically all reasonable type conversions. The only way to change the value of a property is through the property's set method. When a property is part of a BIO, the property's own set-method is never exposed to the client. Instead, the BIO publishes its own set method, which takes the property name and value as input parameters. Because set is the only way to change the value of a property, the BIO and property checks validation rules before allowing the set to proceed. If the set would violate any of the rules, nothing happens and the set returns failure to the client.

Every property has two sets of rules.

First are the style rules. These are the minimum rules that must be satisfied for the property's value to make sense. For example, an integer cannot have a decimal point and a string cannot contain a NULL. Style rules are defined by the transaction server and cannot be changed, removed or augmented.

Next are the validation rules. These are rules added to a property by a business analyst to ensure the data in the property makes business sense. For example, one might require that a currency property used to store salaries must be greater than zero. One might also require that a time property used to store a "paid on date" field must be in the past. The transaction server enforces but does not define validation rules; they are defined by the business analyst.

Potential Rule-Set Lockouts

When a BIO has a rule that checks property P2 whenever property P1 is set, a potential lockout condition can occur. Setting P1 may invalidate the Rule, which would block the set on P1 before a subsequent set on P2 could be invoked. To solve the lockout, both P1 and P2 must be set as a single atomic operation.

This is why the BIO supports multiple property sets in a single set method, called a multiset. The set method accepts an optional number of parameters. Specifically, the syntax is:

```
        set([in] string propName1, [in] VARIANT propVal1,
{[in] propName2, [in] VARIANT propVal2}).
```

During a multiset, all pertinent rules are evaluated using the proposed property values together. If any rules block the set, NONE of the properties are changed. Otherwise (every rule says OK), ALL the properties are changed.

BIO - Property relationships

When a BIO has a property of a an elemental data type (such as string or integer), creating the property is quite simple. Just get the property from the Property Factory. The latter will load and set any validation rules that have been planned for this Property.

Identifying the complex properties of a BIO is a bit more complex because these properties have a metclass as well as a basic type. For example, a property of the basic type Iunknown might be a BIO or a Bag. Further, the BIO might be a customer, and account, or any other valid BIO defined in meta-data.

Currently, the complex types that are directly supported are:

- BIO
- List
- Bag
- Matrix

Each is described in meta-data as follows:

BIO

The property's datatype will be "BIO". The optional default value will be a string of the form:

ClassName[.InstanceName]

Where ClassName specifies the metaclass of the BIO the property represents, and the optional InstanceName suffix is the instance name the initial data that should be loaded into the BIO. If InstanceName is specified, the BIO must be persistent - that is, it must have a meta-data-defined mapper.

List

The property's datatype will be "list".

Bag

The property's datatype will be "bag". The optional default value will be a string of the form:

BaseClass.MetaClass

Where BaseClass is the base class of the objects the bag will contain. This might be "BIO," "property," or any other valid base class. Currently, only "BIO" is supported. MetaClass is the metaclass of the BaseClass. This can be any meta-data defined metaclass.

Matrix

The property's datatype will be "Matrix". The optional default value will be a string of the form:

ClassName[.InstanceName]

Where ClassName specifies the metaclass of the Matrix the property represents, and the optional InstanceName suffix is the instance name the initial data that should be loaded into the Matrix. If InstanceName is specified, the Matrix must be persistent - that is, it must have a meta-data-defined Content Supplier Agent (CSA).

Example 1: Parent BIO with a One-to-One Relationship with a Child BIO

Suppose we have a "foo" BIO with a 1 - 1 relationship with a "bar" BIO. BIO "foo" will have a property of type "BIO". This property will have a default value of "bar". At run time, when the "foo" BIO is created, the system will create a "bar" BIO and make it a property of "foo".

Example 2: Parent BIO with One-to-Many Relationship with a Set of Child BIOs

Suppose we have a "foo" BIO with a 1 - M relationship with a set of "bar" BIOs. BIO "foo" will have a property of type "Matrix". This property will have a meta-class of "Matrix.bar". At run time, when the "foo" BIO is created, the system will create a Matrix and make it a property of "foo". New "bar" BIOs can be created and added to the "foo" BIO's Matrix.

For performance and scalability reasons - and to allow infinitely deep reference cycles - all complex properties are created just-in-time, on demand. Thus, the Matrix property just described would not actually be created when the BIO was created. The first time this property was touched - via either "set" or "get" - it would be created automatically, on-the-fly.

Scalability of Business Information

Intuitively, one would expect a Workflow (BW) to initialize and load a PBIO with data, calling get and set on the PBIO as the BW runs. Unfortunately, life is not this simple. Why not? The first set on the PBIO would start a transaction that would not be complete until persist was called. Thus during the entire lifetime of the BW, the PBIO object would have to occupy space on the server. Multiply this by the number of PBIOs in each process and the number of users on the system and you have a scalability nightmare.

Property Gathering

One solution is to use more lightweight, non-persistent data objects - Context variables - to store data while the process is running. Near the end of the BW, when the needed data has been collected and verified, instantiate the pertinent PBIOs, load them up with information gathered from the BIOs or Context variables, and immediately persist them.

To do this, every PBIO has a gather method in its public interface. This method does the following for each property in the PBIO's property list:

- Search for a variable in the Context with the same name and compatible type.
- If found, set the value of the property in the PBIO to that of the Context variable.

Since every BIO exposes the isDirty interface for its properties, any BW that cares to do so can check which properties were changed by the gather method.

This approach has some potential limitations including:

Namespace chaos: variable naming collisions may impair object reuse.

Gathering complexity: how to enable BIOs to find properties in the context is a non trivial problem

Polluted context: the context would become a garbage can polluted with all the variables needed for any BIO used in the process.

Rendering -rendering information to the channel in a coherent manner is extremely difficult when the data is not organized into BIOs.

PBIO Wrapping

To maintain scalability of business data, we must create PBIOs as late as possible and release them as early as possible. PBIO wrapping does this while bypassing the potential problems of the gathering approach.

A PBIO is just a BIO with an extended interface and a persistence map. Every PBIO object aggregates a BIO and tacks on these new items. Thus, every PBIO defined in meta-data can be instantiated as a BIO, as long as the persistence map and extended interface are ignored.

When a process runs, it may instantiate the Business Information objects it needs in the form of non-persistent BIOs. These objects are lightweight compared to their corresponding PBIOs, since they do not have an extended interface or a mapping layer. Later on, if the process decides to retrieve or persist data to or from the BIO, it instantiates a PBIO to do this. So far, not much different from property gathering.

The difference comes in what the client does to its new PBIO. Instead of using the standard init() to initialize the PBIO, the client instead tells the PBIO to wrap the BIO that is already in the context. "Wrapping" entails the following:

- The PBIO sets its aggregated BIO pointer to point to the BIO.
- The PBIO queries the BIO to get its ClassName
- The PBIO uses the ClassName to instantiate the appropriate mapper object
- Finally, the client tells the PBIO to perform the persistence operation (retrieve, persist or delete), then tells the PBIO to die(). When the PBIO dies, it destroys its mapper, leaves the BIO intact and returns to the client a reference to the BIO.

<u>Persistence Map</u>

The persistence map (PBDMap) applies only to PBIOs. Since BIOs are not persistent they don't have persistence maps. The PBDMap of a PBIO determines where the PBIO's data lives in persistent storage, and describes how to push and pull the data to and from the data store and the PBIO.

The PBIO itself does not know how it is made persistent. All the work is delegated to the PBDMap. The PBDMap can do anything it wants, from pushing and pulling data from a set of relational databases, to dumping a bunch of binary data into a disk file. In addition, the PBDMap doesn't have to map all the properties in the PBIO. The only properties that will be mapped are the ones that are assigned a location by the PBIO's map. Thus, not only does the PBIO not know how it's made persistent, it doesn't even know which of its properties are persistent.

When applications are ported to legacy persistence systems, such as Scopus or SAP, all that needs to be done is to create new PBDMap objects that map to differently organized persistent stores. Everything above this, including the entire representational (PBIO and BIO) model, and the entire behavioral (Process/Step/Operation) model remains untouched and unchanged.

Persistence Methodology

Observe that the interface for PBIO contains only a single persistence function: persist(). When its persist() method is invoked, how does a PBIO know whether to perform an insert or an update?

First, the PBIO knows whether it its retrieve() method was invoked. If not, any information the PBIO contains did not come directly from persistent storage; thus the PBIO contains new data and the persist() must be an insert.

If the PBIO's retrieve() method was previously invoked, then the PBIO contains a (possibly modified) live version of data that came from persistence storage. Thus the persist() must be an update.

In either case, the only way to change any of the PBIO's properties is to use the set method. Thus the PBIO knows which properties have been modified, hence which properties must be pushed (either inserted or updated) into persistent storage.

BIO Instance Management

Every time a BIO is instantiated a new BIO is created. The BIO Factory will never reuse a previously instantiated instance of the same BIO to satisfy the request. Thus, BIOs are never shared; they are inherent per-instance objects.

Behavioral Model

Overview

Using objects to model behavior (e.g. business workflow) was and is the holy grail of true object oriented analysis and design. However, it would be an understatement to say that few systems have achieved this goal. There currently does not exist a single shipping product in market space that uses data-driven behavioral modeling. The behavioral models are one of the most unique aspects of architecture. This section describes our behavioral model.

Workflow

Business Workflow (BW) is the primary abstraction and heart of the Architecture. It represents the highest level definition and behavior of the system.

A Business Workflow, viewed retrospectively, is a series of Steps that occur sequentially as a result of Business Rules. Because the Steps are linked dynamically, the static definition of a BW cannot include all the steps - they aren't known until the BW is running. Rather, a BW identifies only its initial, required Steps (its Backbone). As each Step completes execution, that Step may queue additional Steps. Whether it does depends on Business Rules in the Workflow.

Some Workflows are simple, such as a single Step or a highly preconfigured set of statically linked Steps similar to a Microsoft Wizard. A simple BW might ask for seven specific pieces of information one by one from a user via a GUI, validate the data, confirm it with the user, and then store it.

Other Workflows may be highly variable, with a large network of possibilities branching from a common starting point, with branching decisions determined by a number of dynamically evaluated Business Rules. As a result, trying to draw a BW as a tree or as a network of nodes (both of which would be reasonably accurate) would create more confusion than value. It would be like trying to draw a tree of every possible thing that you, the reader, could do at this moment: virtually impossible, and unnecessary.

Workflows can nest each other. A BW is polymorphic to a Step, so when a BW invokes a Step, it may actually be invoking an sub-process.

During the course of a Workflow, data is collected into a Context.

Most Workflows will instantiate and modify Business Information (BIO), but some BWs may not use any BIOs at all.

<u>Backtracking a Workflow</u>

Workflows can be backtracked Step by Step or all at once back to the beginning, under certain circumstances. The definition of "backtracked" is that:

Corresponding "counter-methods" are called for all BIO methods that were called during the BW.

<u>Steps are backtracked: the Context is reverted one Step at a time</u>

As a result, all affected BIO states and data will be restored as they were before

A typical application of a single-Step backtrack would be a Back button in a Wizard-style interface.

It will never be possible to guarantee the precise reversal of all work done during the course of the BW. This is because Peripherals may take irreversible operations in the outside world, e.g. print an invoice and mail it. However, frequently these operations can be countered with other operations, e.g. print a credit and mail it. It is for this reason that counter-methods are so important.

The Operations taken during BW Steps automatically inform the Context of just how reversible they are (reversibility is a property of an Operation object). Here are the different levels:

<u>Perfectly Reversible: Completely reversible.</u>

Effectively Reversible: The BW should normally consider it reversible, but in fact it is either not 100% reversible, or it doesn't matter whether it is reversed. It is up to the BW to decide whether this is good enough; in most cases it will be.

<u>Irreversible: Renders the entire BW irreversible from this point backward.</u>

Committing a transaction does not necessarily render the BW irreversible, though in most cases it will.

The implication of the ability to backtrack a BW is that the Context must store every piece of information required to do this. In many cases, this is simply remembering which objects changed and deferring the undoing to those objects.

Let X and Y be two different Workflows. If a BIO involved in BW X has been changed by BW Y in the meantime, BW X can not be backtracked.

<u>Workflow Details</u>

<u>Statelessness and Scalability</u>

A BW object of a given metatype, even while running, is stateless. In addition, the Process related objects - Steps, Operations, Actions and Sequences - are also stateless. This greatly increases the scalability of the system by allowing the same instance of the same BW object can be used by multiple different independent threads of execution. To achieve statelessness, the context of each BW is stored in a separate Context object. The Context and BW objects are described in detail in the Architecture Requirements document.

Workflow Context

A BW is always started by the Synchronizer and provided a Context at startup. The Context may or may not have information in it. If it does, it may contain information read from a Peripheral, it may be a saved Context from a previous instance of the process, or may contain other information. The Process, depending on how it is designed, may either use or ignore the initial information provided in the Context.

A process does not control the lifetime of its context. It must be designed to query the context if necessary and act accordingly when it starts.

A Process may receive a "suspend" message at any time. When this happens, it must either finish or roll back any transactions in progress, then make its Context persistent, then terminate itself. This may happen synchronously (client calling "suspend" blocks until the process has gracefully terminated) or asynchronously (client calling "suspend" never blocks).

Restoring a Suspended Workflow

A Process' Context stores references to the Steps the process has completed. When a previously suspended process is restarted, it can obtain from its Context the most recently completed Step, and continue where it left off - e.g. fire that Step's Director to continue at the beginning of the Step that was active when the Process was suspended.

Parallel Workflows and Workflow Communication

A Workflow (BW) can spawn a child workflow; this is called a SubProcess. A Subprocess is given a Context of its own, which is destroyed when the SubProcess exits. The SubProcess can still reach the parent Context by explicitly calling GetParentContext() - and any references not resolved in the SubProcess' own context automatically check the parent context - unless those references explicitly use the Local. prefix when referring to variables.

A Subprocess can be launched in either Inline Child, Inline Asynchronous, or Parallel mode. A Parallel child BW is typically used to perform an independent task, and normally uses its own Presentation viewport (such as a separate browser frame or window). Parallel BWs are neither subordinate to one another (hence the child is not automatically destroyed when the parent exits), nor protected from one another (hence the parent can explicitly kill the child and vice versa). Parallel BWs generally limit themselves to read-only and similar non-interfering types of access to each other's resources, but they are not prevented from performing intrusive operations as well unless a particular resource (such as a Context variable) is Locked by another BW. Locks represent a compromise between flexibility and safety; my personal preference is to have "global" variables like the Language setting Unlocked by default until explicitly Locked, and all other variables Locked by default until explicitly Unlocked. If a BW hits a Lock, the operation fails and normal error recovery takes over.

Inline child BWs are used for tasks that are an integral part of the parent BW -- they are fired as Steps in the BW. They enjoy unrestricted access privileges to the Context, Presentation viewport, and other resources of its parent. This resource sharing is made safer by the more predictable nature of the control flow: from the time an Inline child process receives control and until it exits, the parent remains stopped, waiting for it to exit.

Inline Asynchronous child BWs are similar to function calls, in that they receive control immediately and return when they are done. However, this functionality is sometimes insufficient. For example, during the course of the workflow we may discover (perhaps by consulting our OLAP information source) that the user is a good candidate for purchasing a particular product. This discovery may occur at various points within the workflow, and therefore should be modeled as a general business rule rather than integrated into any one step. Such a rule, however, could fire at a time when the workflow is engaged in an important task, and flashing a special offer on the customer screen is inappropriate. Normally, such rules would be fired by the Synchronizer, upon receiving notifications from running BWs. The BWs themselves would merely notify the Synchronizer periodically, neither knowing nor caring about such rules.

New BWs can also be queued automatically by the workflow engine. Such BWs are initially registered as Interrupts by another BW, with conditions under which they must be invoked. The conditions are evaluated by the engine at each step transition using global variables and variables from the context of all running processes to which the Interrupt is Relevant. Relevance is determined through a qualification scheme structurally similar to those used for security credentials. If the conditions are satisfied for a particular process' context, the interrupt step sequence is inserted as an Inline Queued child of this process. The transition priority into the first step of an interrupt sequence is set at time of registration of the interrupt sequence.

Workflow Queueing

Application workflows frequently throw events that are caught by the Synchronizer, which starts child workflows in response. A good example of this is the Navigator, which consists of a list of buttons. Each button, when clicked, throws a different event at the Synchronizer. The Synchronizer has Event Selectors that cause it to start a different child Workflow for each of these Events. This scheme is both powerful and dangerous.

It is powerful because the server "remembers" the state of every workflow, which makes the workflows easier to reuse. Any child workflow can interrupt any parent workflow, and when the child is done the parent continues where it left off. Because the state of a workflow is encapsulated entirely in its Context and completely decoupled from the workflow itself, this is achieved without any requirement for the parent and child to know about each other.

It is dangerous because workflows may stack up in the server to arbitrary levels of depth, allowing a badly designed workflow to force the transaction server to store the state of stagnant workflows that the user will never revisit.

The solution to this dilemma is to provide a mechanism whereby the business analyst can decide, when he needs to run a certain workflow, whether to stack a child workflow as described above, or to have the server switch to this same workflow if it is already waiting somewhere in the stack.

This behavior is orchestrated by the Synchronizer's selectors. The workflow merely throws events as before. The style of the selector that catches the event determines whether a new workflow is always started, or whether a new workflow is started only if no workflow of the same name is not already running in this session. Conceptually, the difference here is similar to that between gosub/return and goto.

Workflow - Step Polymorphism

A Workflow is polymorphic to a Step - in fact, a Workflow is a Step. All Workflows are Steps, but not all Steps will be fired as independent Workflows - though they could be. A Workflow is normally designed as a backbone Step. That is, a Step that consists of a Director and possibly a Rule, but no operation. This Step's Director queues the backbone of the BW.

Starting a Workflow

The section regarding the Synchronizer and Event Machine describes how peripherals and other external objects throw events at the Synchronizer, and how the Synchronizer responds by launching workflows, running operations, evaluating rules, etc.

Related Objects

The Behavioral Model consists of several other objects which fulfill the responsibilities of Business Workflow. These objects are described below.

Steps

Step Overview

A Step is a stationary point in a BW. During a Step, Operations are taken - for example, information may be requested of and received from a GUI, and stored in a data store. A step consists of a Rule, an Operation and a Director. The Rule may block the Step from occurring, the Operation determines what the step does, and the Director determines what Step(s) come(s) next.

A Step references a Director that when executed, provides the next Step(s). The Director may be simple, such as identifying a static, hard-coded next Step. Or the Director may perform a highly complex analysis of the BW history to determine dynamically a sequence of following Steps - with possible interruptions or diversions. This simplifies the architecture by setting the invariant that Directors always determine the next step, while hiding the complexity of determination within the Directors.

Steps are reusable. Steps may be reused across BWs, and therefore a given Step is never to be considered to belong to a single BW. A BW uses steps, but it does not own them.

Step Control Flow

Steps are the unit of dynamic control flow for Workflows. Control flow of a Workflow runs as follows:

Control flow is managed by the Workflow's step stack 400. Every Workflow has a start step or step sequence (e.g., step sequence 410 in Fig. 4). This is pushed onto step stack 400 when the BW starts. After the first Step completes its operation, the Step fires its Director. The Director pushes a step or step sequence (e.g., step sequence 420) onto step stack 400. The Workflow goes to the next Step at the top of step stack 400. This control sequence is depicted in the diagram of Fig. 4.

Backtracking and the StepStack

There are three major types of backtracking that can be performed within a BW: (1) backtracking to a parent step, (2) backtracking to a peer step, (3) backtracking to the nth previously executed step.

Of these three, backtracking to a parent step is the least complex. A step is popped off of the step stack only after all of its children have finished executing. This guarantees that all of the parents of the currently executing step can still be found in the step stack. Furthermore, there will be at most one parent per level and, if present, the parent will always be the first element on that level. Thus, backtracking to a parent step can be implemented by popping levels off of the stack until the desired parent is found in the first position of the top level.

The second and third methods of backtracking are much more complex because of the incomplete history provided by the step stack. The step stack is intended to provide a glimpse of where the BW is headed and, therefore, does not contain any steps that have executed to completion. Given this behavior, backtracking to a peer step and backtracking to the nth previous step can only be accomplished if supplemental information is maintained regarding the history of the BW.

Since only the unexecuted peers of the current step remain in the step stack, backtracking to a peer can only be performed if the step stack maintains a list of previously executed steps for each level of the stack. Backtracking to the nth previous step requires even more information, as it cannot be performed without the entire history of the BW. The reason for this is that the nth previous step S_P may actually be located at a deeper level in the step tree than the current step S_C. Therefore, it is necessary to first backtrack to the parent from which both S_p and S_c were descended. Then, the step stack must be filled, by firing directors, until it reaches the state it was in when S_p was initially executed.

Directors

A Director is a composite object (a tree) consisting of a single method, eval. This method fires the Director's topmost (root) Rule (invokes eval on the Rule). If the Rule returns TRUE, the Director jumps down into that node of the tree, queues any Steps it finds and evaluates any Rules it finds.

Queued vs. Immediate Directors

Normally, the director itself evaluates the Rules that determine whether Steps are queued. But often times it is useful to postpone evaluating a Rule (for example, a Rule might branch based on the outcome of a Step that is only queued, not executed). Using the Workshop, the business analyst may indicate which Directors are to be queued rather than executed. When the system encounters such a Director, the system queues that Director. When the queued director is reused in the tree of another Director, the entire subtree rooted at the queued director is queued.

Ideally, we would like the clients of a Director to be unconcerned with whether the Director is queued or immediate. The goals are summarized in Table 2 below:

Table 2

TYPE	EVAL	POP FROM STACK
Immediate	Fires rule(s), queues step(s)	N/A (Director never gets popped from stack, because it was never pushed onto the stack).
Queued	Pushes itself onto the stack	Fires rule(s), queues step(s). Any queued child Directors are queued, when their parents are fired.

To achieve these goals the director must know when it's being popped off the stack. Since this is inadvisable (in fact, impossible), we must provide a different mechanism. What we end up with is two different methods to activating the Director, eval() and evalNow().

In eval(), the director checks whether it's queued. If it is, it pushes itself onto the stack. If it's not, it evaluates its Rules and (conditionally) pushes its Step(s) onto the stack.

In evalNow(), a non-queued director does the same thing as it does in a normal eval(). A queued director ignores its "queued" flag, evaluates its Rules and (conditionally) pushes its Step(s) onto the stack.

The default is for all directors to be queued. Immediate Directors would be used in very rare circumstances, if at all.

<u>Types of Directors</u>

The framework currently includes two kinds of Directors, and it is designed to be extended to support many more in the future. Consistent with the OO design of the framework, new types of directors can be added to the system with no changes to existing code or meta-data tables. This includes the Director Factory, which will be able to create, cache and manage the state of any new kind of Directors added to the system - with no changes to its existing code.

As of the Beta 2.0 release in February 1999, the framework includes two kinds of Directors - Step Queuers and Step Sequencers.

<u>Step Queuers</u>

A StepQueuer consists of a Rule and a single Step. When the StepQueuer is evaluated, it fires its Rule. If the Rule returns FALSE or fails to execute, the StepQueuer does nothing and returns. If the Rule returns TRUE - or if the StepQueuer does not have a Rule, it pushes its Step onto the Step Stack.

Step Sequencers

A StepSequencer consists of a Rule and an ordered list of Directors. The StepSequencer behaves just like a Queuer - except that where the Queuer pushes a Step onto the Step Stack, the Sequencer visits each of its children, and invokes eval on each, in the proper order. Each of these children can be any kind of Director - a Queuer, another Sequencer, or any new kind of Director may introduce in the future. This continues recursively; there is no arbitrary limit to the depth or complexity of a StepSequencer.

Control Flow and Threading Models

In the simplest case, one thinks of a program running on the same logical thread as its user interface. But since most programs spend the vast majority of their time waiting for the user to do something, this simple case simply wastes potentially huge amounts of processing power. The Architecture puts these CPU cycles to productive use, realizing tremendous scalability improvements. This is done by dividing the Workflow's logical control flow from its physical threads, and by decoupling both from the contact channel logic.

Decoupling Logical Threads from Physical Threads

As mentioned earlier, a Workflow is stateless and maintains a step stack in the context. Thus, at any moment in time, if the Workflow object were destroyed and its Context saved, that Context could be handed to any other Workflow object and continued. This leads to the intriguing realization that the physical thread executing a given Workflow can be destroyed at any time without any loss of information. The context can be handed to any other physical thread in any other Workflow object and continued.

So what do we gain by destroying the physical thread that is executing a given instance of a Workflow? This means the transaction server never has any idle threads. Any operation that would wait for the user to do something (such as fill out a form), fires off the Interactor and - without waiting for the

Interactor to complete - immediately destroys the Workflow's physical thread. While the interactor is running, the transaction server stores only the Context of the Workflow, a static COM object with no physical thread of execution. When the Interactor is complete, the Context is handed to any available Workflow which then continues executing it.

An intuitive sense of what is going on here can be gained by imagining the Workflow as pictured in the Visio notation of the Workshop - a giant tree that executes in left to right depth-first order.

The classical model of control flow enters at the root node, walks depth first down to the leftmost, deepest node, and crawls along the leaf nodes of the tree from left to right. If it is suspended, its execution state is saved (position in the tree, node visitation history, etc.) and later restored for subsequent execution.

In the model of control flow, instead of crawling among the leaves of the Workflow, the focus of control enters at the root and immediately jumps down to the appropriate leaf node - which was saved in the Context at the time the Workflow was previously suspended. Instead of crawling through the Workflow and waiting at certain nodes for Interactors to complete, the focus of execution jumps into the Workflow directly to the pertinent Step, fires an Interactor, sets a pointer in the Context and dies.

<u>How the Threads Interact</u>
<u>Operations</u>
<u>Operation Overview</u>

Operations provide an abstract interface to do things. All Operations provide the same interface, hiding what they actually do from the client objects that use them. Operations enable client objects to do things without knowing what will actually get done. A single Operation may perform a single action. Or it might be a sequence of many other Operations, each of which is simple or itself a sequence. This hides functional complexity from client objects, breaks the dependency between clients that perform operations and the operations themselves, and it builds operations into a hierarchy of reusable, compositable objects.

Operations are Commands that change the state of BIOs. An Operation is an active command, a piece of work performed by the system. An Operation might request data from a Peripheral; it might send data out to a Peripheral; etc. Operations primarily get data from and put data into BIOs.

Operations are atomic, either succeeding or failing (and perhaps rolling back) as a unit. Though an Operation is atomic, it may consist of a hierarchy of actions. If it does, the actions are considered a single atomic unit. They therefore run in a statically defined order with no conditional control flow.

Operations are sequential. Operations are performed in order during the course of each Step.

Operations may be synchronous or asynchronous. Most Operations wait until they are complete before letting the system move on to the next Operation, but an Operation doesn't have to pause the system if there is no reason to do so - if subsequent Operations do not depend on its results.

Operations are reusable. Operations may be reused across Steps, therefore a given Operation is never to be considered to belong to a single BW. Operations exist independently within the Process Manager. A Step uses Operations, but it does not own them.

Because Operations - unlike rules - change the state of BIOs, operations provide an interface to "undo" what they have done. Not all operations are reversible, so it is up to the individual operations how to implement this interface. The undo interface facilitates backtracking BWs; see the section above on backtracking a BW.

Business Rules are designed to prevent Operations. It is the job of Business Rules to determine whether Operations should be allowed to perform their functions. When an Operation is executed, the Process Manager checks whether any Business Rules in the system should be "interested" based on the Operation and the Context. Any relevant Business Rules will be checked, with a single failure causing the Operation to fail and return an explanation.

Additionally, an Operation has its own private Rule, used as an "inhibitor". The Operation Rules do not "know" about operations.

Operation Details

Operations perform everything that actually "happens" in a workflow. The interface to an operation consists primarily of doo ("do" is a reserved word) and undo. Every operation has a rule, called an inhibitor. Upon receiving a doo message the operation fires its rule. If the rule returns FALSE, the operation returns without doing anything. Otherwise, the operation executes and returns SUCCESS or FAILURE depending on the result.

An operation is a composite command pattern consisting of either a sequence or any of many possible different action objects, both of which aggregate an OperationImp object to handle context logging. An action is a single independent exploit. A sequence is an ordered list of operations, each of which is itself either an action or another sequence. Thus an operation may be as simple as a single action, or a complex multiplicity of actions arranged in an ordered tree. Regardless of the particular case, it presents a simple doo, undo interface to the client.

Transactions and Execution Semantics

Every operation has a rule that can block it, and if the operation runs it may succeed or fail to perform its given function. The most interesting case is what happens when the rule allows the operation to run, but the latter fails to complete its given function. What happens in this case depends on whether the operation is a transaction.

If the operation is a transaction and it fails, it will roll itself back and return failure. If it is not a transaction and it fails, it will simply stop executing and return failure.

This leads to five possible results of an operation, only one of which indicates that the operation ran successfully. The results are summarized in Fig. 12.

In the flow diagram of Fig. 12, an operation is first triggered in stage 1210. One or more rules are then evaluated. If all rules return a TRUE value, the operation is executed in stage 1220. Otherwise the operation fails and return a BLOCKED value in stage 1230. If the execution of the operation is successful, a SUCCESS value is returned in stage 1240. Otherwise, stage 1250 determines whether the operation is a transaction, in which case the operation proceeds to stage 1260. Otherwise, the operation fails and a DIRTYFAIL code is returned in stage 1270. Stage 1270 determines whether the operation can be rolled back, in which case a CLEANFAIL value is returned in stage 1280. Otherwise, a CRITICALFAIL value is returned in stage 1290.

Most clients would consider anything but SUCCESS a failure. They are not concerned with the subtleties of failure including BLOCKED vs. DIRTYFAIL vs. CLEANFAIL vs. CRITICALFAIL.

One immediate exception to this is Attendant objects - when an operation is used as a validation mechanism on a BIO or a Property. Here the operation's return values are interpreted according to Table 3 below.

Table 3

Operation	Validation	Explanation
BLOCKED	OK	Rule confirmed valid data, no action necessary
SUCCESS	OK	Rule invalidated data, operation succeeded

		(fixed the bad data)
DIRTYFAIL CLEANFAIL	BLOCK	Rule invalidated data, operation failed to fix it
CRITICALFAIL	BLOCK	Rule invalidated data, operation failed to fix it, some data may have been corrupted.

Triggers, Mini-Operations and Spheres of Control

In many cases it is useful to restrict what an operation can do, in order to guarantee the undoability of a larger transaction in which the operation plays a role. For example, when an operation is fired as a trigger from a set on a BIO, that operation is not allowed to perform any action which could jeopardize the undoability of the set command.

One could go through great effort to provide a limited API of "restricted" operations appropriate to different spheres of control. However, a far more elegant solution is to allow any operation, but restrict the universe in which it lives. If it is not immediately clear why this is more elegant, recall that in any object oriented system there is no global state. The universe in which any object operates is merely what is passed to it. The standard IOperation interface calls for an entire Context object. Thus, standard operations can use the entire context. However, if we define new interfaces similar to IOperation we can set up these new interfaces so that call for sets of information more limited than an entire Context. Thus, any real Operation object could be used, but if it will simply fail if tries to use anything that does not exist in the universe handed to it.

Example:

The normal IOperation interface has a "do" method that takes the form:

```
        HRESULT doo([in] IContext *theContext, [out,
retval] OpResult *theResult);
```

(note that "do" has been intentionally misspelled as "doo" to avoid reserved word conflicts)

Thus anything the operation wants to see, it must get out of the Context that is provided as an input parameter. The context (or universe) of this operation is the Context. If we make a new interface, say IBDTrigger with a "do" method as follows:

```
        HRESULT doo([in] IBD *theContext, [out, retval]
OpResult *theResult);
```

Then anything the operation wants to see, it must get out of the BIO that is provided to it. The context (or universe) of this object is a BIO.

What makes this approach particularly elegant is that when a particular operation can be made to work in both universes (context and BIO), it merely supports both interfaces and takes action as appropriate, depending on through which interface it received a "doo" message.

Spheres of Control

The spheres of control summarized in Table 4 below are supported by the framework.

Table 4

Sphere	Interface	Explanation
CONTEXT	Ioperation	Used for general operations fired from Steps
BIO	IBDTrigger	Used for BIO and Property triggers.

Operation Meta-data

Every operation is either a sequence or an action.

An action is a COM object that implements the IOperation interface and does a single, atomic thing. There are many different action objects, each identified by a meta-data ClassName that maps to the object's COM class ID. Different action objects are real COM objects, each with a different COM class ID. Thus, to store an action in meta-data, one must merely store its COM class ID.

Like an action, a sequence is a COM object that implements the IOperation interface. But the similarity ends here, as a sequence doesn't actually do anything, in the sense that actions do things. A sequence maintains a list of child operations, each of which might be an action or another sequence - the parent sequence doesn't know (or care) which. When a sequence receives the doo message, it delegates it to its children (in forward order). A sequence also delegates the undo message to its children - in reverse order. There is only one sequence COM class; at run time, it gains a className and a list of child operations when a client sends it the init message. Thus, to store a sequence in meta-data one must store a list consisting of the names of its child operations.

The IOperation interface is flexible, as it takes a variable number of arguments. So every operation stored in meta-data - whether an action or a sequence - must list the parameters it requires. All parameters are passed as VARIANTs and are specified as [in,out]. The caller is responsible for allocating and deallocating them. The operation receives them and may modify them.

<u>Contact Channel Model</u>

While the design philosophy of the present invention emphasizes the relative unimportance of Presentation relative to the prominence of Workflows, we obviously need to provide Presentation since over 90% of applications require it. The Conversationalist Presentation is as thin as possible, serving primarily to display information, present prompts and receive data from users.

Opposite the functioning of most application frameworks, the Workflow drives Presentation. This enables an organization to use a single BW as a canonical model that drives many dissimilar interfaces. BWs determine the order in which

Presentation delivers and receives information from the user, as they do with any Peripheral. Interaction between a BW and a Peripheral (such as the GUI) is very much like a conversation, hence the name Conversationalist.

Presentation responds to the direction of BWs regarding the order of activities, while having influence over the flow by offering users "trigger points" such as Next, Back, Suspend and Cancel buttons which can alter the BW flow from Step to Step. In addition, like other types of Peripherals, Presentation can trigger a BW without the intervention or even the knowledge of the user.

The Conversationalist will be primarily web-browser-based. It will be designed to present information and sets of prompts in several styles, e.g. Wizard, Line by Line, and Visual Story (a rendition of all or part of a BIO as a story on the screen through which a user can browse).

Identification

One of the critical tasks any contact channel must do is to identify the users of the system and ensure that server transactions invoked on their behalf execute at the appropriate security level.

Every contact channel identifies users in different ways - phone numbers for CTI, source or destination URLs or cookies for web, "to/from" addresses or embedded tags for email, etc. But framework, in which workflows are decoupled from channels, requires a channel-independent way of identifying users.

Additionally, in some cases the particular user involved is not important, but his origin may be. For example, a web site exposed to the internet might allow any user to hit the server and browse an on-line catalog, while automatically identifying repeat users and providing them a customized experience. Then, if the customer decides to buy something, the site would switch to a secure SHTTP protocol and verify the user's identity.

Levels of Identification

This example illustrates three levels of identification:

- Anonymous guest - no origin or user known; non-secure and non-customizable.
- Origin ID - origin known, user unknown; non-secure, but customizable.
- User ID - origin may or may not be know, but user is known and is secure - secure and customizable.

Each of these identification levels are discussed in detail below.

Anonymous

This is the default identification level used when a user hits the server, but the channel is unable to determine who the user is, or where he is coming from. Some servers, such as Financial institutions) may complete reject such access. Others, such as on-line stores, may accept and encourage such access, but limit the server transactions such users can invoke.

When an anonymous user hits the server for the first time, the channel may place Origin ID information into that user's local site. This information could be used to identify and welcome the user, and customize his experience, the next time he returns to the site.

Origin ID

Origin ID is a non-secure method for casually identifying users without using the overhead and inconvenience of an explicit, secure login. This is convenient for welcoming users and providing a customized experience that does not include personal, financial or other information of a secure nature.

A single user might have multiple origin IDs. This would certainly be the case if the same person used the service under different roles, for example when an employee purchases something from his employer's web store. The reverse can also occur: many different users might share the same origin ID.

This would happen, for example, when a husband and wife both shop from home using the same browser and computer, but each uses a different credit card when purchasing on the web.

User ID

User ID is secure method for identifying an individual user. Unlike other, more casual forms of identification, user ID can only enter the server through a secure mechanism - for example a secure SHTTP web page, a PGP encrypted email, or an encrypted GSM digital cellular phone.

Identification Resolver

Though the above requirements may sound simple, they require a lot of careful and complex sever functionality. Framework includes an Identification Resolver component which carries out these duties. Specifically, the Resolver performs the following tasks:

- Converts channel-specific IDs to channel-invariant IDs, and vice-versa
- Generates server unique identifiers in flexible formats

Interface Components

Standard Browser Functionality

The applications will inherit the standard browser interface components such as the menus and button bars.

Tree/Menu

The left quarter of the screen will be occupied by navigation controls. The look and feel of these controls will be highly customizable but they will be designed to take one of two commonly seen forms: Tree or Menu. A Menu will consist of a series of text prompts or buttons, arranged vertically. A Tree will consist of a series of text items (often decorated graphically) arranged in a typical hierarchical structure. The advantage of a tree is that it shows the user where she has been and what her options are from this point on.

Views

A View is a set of visual objects to be presented together. This may consist of nothing but a designation of prompts and field descriptions (GUI type, data type, length, formatting, etc.) which the Conversationalist GUI draws and arranges. Or it may be a highly detailed page or frame with a precise visual object layout. The Conversationalist GUI can communicate with JavaScript and ActiveX components, and in fact a View may contain nothing but a single ActiveX editable spreadsheet control from which all the data required by the BW Step is derived.

In BW terms, one View will ultimately exist for each BW Step in which Presentation interaction occurs. A single Step should not call for multiple interactions with the same Presentation; the designer should use multiple Steps in a BW to determine the order in which the Views should be presented. The presentation of Views and the user's interaction with the Views is referred to as a Conversation.

Presentation supports three kinds of views:

- No View. First of all, it is optional for a designer to manually construct a View for a particular BW Step interaction. If this is not done, the Conversationalist GUI will display the prompts passed from the Step in a default fashion. This default Prompt Set Presentation Style will be partly determined on the user's preferences and partly determined by preferences set in the Context during earlier Steps in the BW (see #2 below).
- View Properties Only. If the designer builds a View to correspond with a given BW Step, his simplest design option is to set View properties only. This translates to setting preferences in the Context which can either operate only within the current View or which can be left in place for future Views until overridden. Properties include Prompt Set Presentation Style (either Wizard or Scrolling Script) and Background

Image (an easy way to complete a Wizard-style View without resorting to HTML as below).

- Custom View. The designer may choose to write HTML or use web page design software to lay out the View in highly specific detail. The resulting page (or frame) may include ActiveX components, JavaScript, or any other constructs typical in a web page. Ultimately this page needs to provide responses to the prompts passed from the calling BW Step. It can do this by employing specialty tags in the same way an auto-generated View does.

The View can never decide what is to be displayed. The driving BW Step determines what information is pertinent and the View decides merely how this information will be displayed and acquired.

A View can either set its Prompt Set Presentation Style or inherit (from the current Context) the one used by the last View. Styles are:

- Wizard: A series of frames with one Prompt Set (often just one prompt) displayed per page, often with a specialized background image.
- Scrolling Script: Each new set of prompts and response blanks appears on a new line.
- See the section on Conversations below for information on how Views fit together.

Conversations

A Conversation is the user's interactive experience with the user interface. Because most Conversations in the Demand Chain Product Line will take place through standard browser-based graphical user interface Peripheral, this Peripheral is called the Conversationalist GUI.

As a user works his way through an application by clicking around an Extranet or Internet web site, the application's BWs tell Presentation just enough information to get by. At a given Step, assuming that Step requires interaction with the Conversationalist GUI, the BW sends the Conversationalist Peripheral the following information:

- Identifier of the current Step. This is what the Conversationalist GUI uses to look up whether it has a stored View for that Step.
- A Prompt Set. This consists of field names that the Step expects the GUI to fulfill.

As described in the Views section above, each View is free to determine its own Style or to inherit the Style from the current Context. In most cases the Views will be presented in the same Style, although this is not required.

Contact Channel Modeling Details

A business workflow (BW) communicates with the outside world through delivery channels. A BW Step (or, more precisely, an Action within an Operation within a Step) orders information transfers out of the BW Context into a delivery channel and inputs from a delivery channel into the Context. An Interactor is the unit of such data transport.

Interestingly, Interactors enable Workflows and delivery channels to communicate without knowing about each other. It is crucial to ensure that neither becomes polluted with details or knowledge about the other, so that each remains independent and reusable. This is how a single process can be delivered across multiple channels, and how a single Interactor can be used in many different processes.

Interactors as seen by the Workflow

An Interactor is not itself a BW, in the sense that it is not composed of Steps, Operations, and so on. However, from the point of view of control flow and data sharing the Interactor is similar to a Subprocess. An Inline Interactor runs as a simple function call and, when done, returns to the Action that fired it. A Tearoff, or Parallel, Interactor executes asynchronously and independently from the parent BW, but shares the parent's Context subject to the data locking mechanism. A BW can spawn and maintain multiple concurrently active Interactors.

An Interactor cannot directly communicate to another Interactor, nor back to its parent Workflow, except to signal its own completion. Rather, an Interactor performs its task by presenting the values of specific Context variables and storing user input in Context variables. The particular variable to which an Interactor binds is supplied to it by the Action at the time the Interactor is invoked. If the value of the variable is an aggregate data structure such as a Business Information Object (BIO), the Interactor must be designed to render data aggregates of this type. In this case, the Interactor's appearance and the binding of its elements to data elements within the aggregate is defined at tool time through a Glyph (see below).

An Interactor may publish and subscribe to system events through the Synchronizer, and events may also fire from triggers linked to the Context operations the Interactor carries out. Synchronizer events are the sole mechanism through which Interactors can perform extra-Context activities, and through which they may receive additional instructions from BW while in progress.

Control Flow and Threading Models

Unlike many systems, including Microsoft Windows, the engine is not driven by its user interface. It is useful to think of the server as executing on two sets of threads: Workflow (BW) threads and user interface (UI) threads. UI threads are typically created by the operating system or a web server to process UI events such as button clicks. BW threads are created by the Synchronizer component within the engine to run Workflows. If the task is initiated by the UI (a web hit, for example), the UI thread handling the request notifies the Synchronizer, which launches a BW thread. If the task is initiated by BW, the BW thread notifies the manager for the desired Contact Channel (C2), which causes the system to launch a UI thread.

The Contact Channel Manager (C2M) never makes BW threads wait for UI threads. When an Action within a BW "drops" an Interactor into a Contact Channel, the C2M returns control to the Action as soon as basic initialization is completed. The C2M then uses UI threads to transmit information to the user, to manipulate Context variables in response to Presentation events, and ultimately to notify the Synchronizer that the Interactor has finished work. In the meantime, the BW thread may either block, or, in the case of a Tearoff Interactor, continue with the Workflow. The engine may also be able to reuse the physical BW thread while the Workflow is idle. With care, it is even possible to design the system to begin a Workflow on the same UI thread that notifies the Synchronizer of session start, and return this UI thread to C2M after the Workflow drops an Inline Interactor and thereby places itself into a wait state. The details of this scheme given in the section describing Workflow.

The C2M generally does not block UI threads, either. A UI thread created or allocated by the operating system to handle a UI event is used to manipulate Context variables, to formulate a response for into the channel, and for such other tasks as may be necessary to handle the event. Sometimes, however, it is necessary for UI threads to wait for BW threads. For example, a session's first UI thread is not caused by a UI event in any Interactor, but rather is used as a vehicle to deploy the Interactors in the first place. Since the Workflow may wish to deploy multiple Interactors at once, the C2M blocks this UI thread until the Workflow gives it a go-ahead to deploy. The same condition can also occur at a later time after a blocking Interactor quits and the UI thread that had brought the quit message is the only one available to deploy replacement Interactor(s). To make this deployment possible, C2M will block the UI thread until another go-ahead signal is received from the Workflow.

Viewports

The Workflow has limited control over the way in which Interactors are rendered. This is exercised by assigning the Interactor to a specific Viewport. In a GUI channel, Viewports typically correspond to windows or browser frames. The Interactor can be placed into a Viewport in Erase, Overlay, or Append mode. Erase causes the previous Interactor in the Viewport to immediately exit and return to its parent BW if it has not already done so; Overlay puts the previous tenant in a dormant state, but allows it to reclaim the Viewport after the newcomer quits; and Append places the new Interactor at the bottom of the Viewport following any Interactors already there. (The Append behavior is needed to implement script interface: "Please give your name?..." – "Joe Bloe" – "Social Security Number?..." – "123-45-6789," etc. Each question is a separate Interactor – indeed, a separate Step – because the Workflow must be able to branch based on the answers.) On non-GUI channels, Viewports could be given a logically similar interpretation, or ignored.

Operations on the Context

Form-like renditions of simple properties and variables normally make all the necessary Get calls at the beginning and all the necessary Set calls at the end of their execution. (If an Interactor must deliver multiple properties into the same BIO, it groups them together using the Multiple Set method to simplify validation and avoid validation interlocks.) Certain types of Interactors, including those designed to render a tree or an Active X control, may require live exchange with the user and the Context. Such complex controls must map onto similarly complex objects within the Context that support the interface necessary to transfer the data to and from the control and to carry out the interactive operations supported by the control. At the same time, this complex object must usually remain accessible to the BW, so that its data may be populated, analyzed, persisted, etc. It may be a good idea to implement this special object as a special type of Bag of BIOs.

Each Context object has the ultimate responsibility for maintaining its internal integrity by enforcing its validation rules. However, an Interactor may publish a list of business rules that it guarantees to hold True at all times when it calls Context object methods. Any rule from the reusable pool of business rules may be published in this way. If a Context object must later evaluate one of these rules for validation purposes, it has the option to instead trust the Interactor and skip the rule. This feature improves system performance when the client computer is sufficiently powerful to run the Interactor code. Alternately, a separate and single piece of code that evaluates the rules could be used. This code could be moved from the transaction server to the client, as needed. For various reasons, including the importance of a particular business rule, the objects may choose not to trust the Interactor's guarantees.

If any of the Context object methods invoked by the Interactor returns failure, the failed method generates an Exception Object (analogous to a C++ Exception object). The purpose of the Exception Object is to facilitate interactive resolution of the problem by the Interactor. The Interactor may then retry the failed method with different data and/or invoke other methods on Context objects. For example, if a Set fails due to a validation rule, the Exception Object might include an error message and a list of properties constrained by the rule. The Interactor's default behavior might be to maximize its window, report the error message, and ask the user to re-enter the properties involved. The Exception Object may identify to the Interactor additional Context objects and thus enable the Interactor to access these objects. If error handling is unsuccessful, the Interactor may choose to exit with an error return code. The return code reaches the parent BW only in the case of a synchronously called Interactor.

The Interactor's fail is Dirty in most cases, since it does not attempt to reverse the successful method calls it has made. The Interactor's Undo may attempt to reverse these calls, if the Interactor supports Undo. Note that Undo's are problematic in an environment with multiple concurrent threads over the same Context, because even simple Set methods may not be reversible. In all cases, Undo would remove the Interactor from the hosting Viewport.

Glyphs

A Glyph is an atomic component of an Interactor. The function of a Glyph is generally to render a single Context variable or BIO property, called Bound Variable. Glyphs can be aggregated into a group, which is itself polymorphic to a Glyph. Each node and each link of the Glyph tree includes a list of named attributes with string constant values. Examples of attributes might be "IvrPrompt," "GuiWidgetType," "Color," "FontSize," "LabelText," "LabelFontSize," etc. On rare occasions the value of a Glyph attribute may be a binary object (blob).

An attribute specified in a parent Glyph or on a link can have one of the following effects on child Glyphs:

- Suggest: The value specified higher in the tree provides default for child Glyphs.
- Inhibit: The value logically AND's with child values. This is useful for control of visibility, enable status, etc.
- None: The value does not propagate into child Glyphs. Attributes like label text or pixel coordinates are examples.

The inheritance model is fixed for each attribute, i.e., "Visible" is always Inhibit, "Color" is always "Suggest," etc. It may be a good idea to hard code these choices, because changing any of them would unpredictably break previously designed Glyphs. If the code cannot determine the inheritance model for a particular attribute name (this only happens when running against an older version of the meta-data), no inheritance is assumed. All top-level Glyphs are treated as aggregated into the systemwide "Mother of all Glyphs" for inheritance purposes.

The Glyph tree is interpreted by a contact channel specific Composer component, which constructs the output stream appropriate for the channel. In an interactive channel, any response produced by the channel is directed back to the same Composer. The Composer may react to these user events by writing data into the Context and by firing Synchronizer events.

The Composer supplies default behavior for any omitted Glyph attributes, and ignores irrelevant attributes and those it cannot recognize. In most cases, only a small fraction of all supported attributes would have to be given explicitly within the Glyph. Even without any attributes present, a reasonable rendition should be possible based on property style and other descriptive information available from the bound variable. New attributes may be added in later versions of the product to support new functionality (say, a new GUI widget).

One attribute of a composite Glyph is its Layout Policy. On a GUI channel, the policy determines screen positioning of child Glyphs within the parent Glyph window. Standard layout policies include "Row," "Column," and "Tab". (In the latter case, child Glyphs share the screen area of the parent inside a tab control.) New policies may be added if the Interactor code supports them. Since the Glyph hierarchy typically reflects semantic relationships between data items, it may also be helpful on a non-GUI channel, for example, to determine the order of input.

Glyphs and their attributes are created at tool time and become part of the meta-data. Top level Glyphs are named, and make up a Glyph Library. The designer can construct new Glyphs at tool time that would aggregate other named Glyphs. In meta-data, therefore, the Glyph parent-child is a many-to-many relationship, so that, say, an "Address" Glyph may be aggregated by multiple forms. This is useful when different Address Glyphs must be designed for different languages, tenants, etc. As reconstructed by the factories at runtime, however, each composer effectively works in a separate Glyph tree and can make changes to attribute values visible only to itself. The Composer begins with a top level Glyph of that tree, and follows down its hierarchical structure to do the rendering.

<u>Glyph-to-Data Binding</u>

Bound variable name of a Glyph can be thought of as a Glyph attribute with a unique inheritance model. Specifically, any value attached to the link always overrides that of the node, and if the value begins with an @ sign, the value from the parent node is PREFIXED, otherwise it is ignored. The name is returned as a string and is resolved against the Context by the caller (normally a Composer).

There is tension between the role of the Glyph structure as the definition of screen layout, and its role as the definition of the mapping between fields and data elements. This dual interpretation of the Glyph tree facilitates reuse when the visual presentation and the logical structure of the data mirror one another, but creates difficulty when they do not.

Our treatment of bound variables is aimed at relieving this tension at least somewhat. First, a Glyph, and particularly a named Glyph out of the Glyph library, can be handed a "root" property through the link by its parent and render its sub-elements according to its own design. Hence, say, an Address Glyph can be reused on any Address BIO no matter what larger data structure that Address may be aggregated into. At the same time, a Glyph can also go outside the root BIO, so that one can show, for example, time of day at the corner of a form. At the extreme, one can program all or most of the Glyphs in a tree with fully qualified variable names, to obtain a data mapping that is independent of the screen mapping, but that is also not reusable since it is meta-data-determined. Finally, our model makes sure that the data mapping given by the parent to the child through a link attribute is more than merely a suggestion that can be immediately ignored. This makes Glyph structure design a bit less error-prone, even though the mapping can still be ignored at the grandchild level.

The data binding logic within a Glyph is not visible to its run-time clients. From their perspective, only terminal (i.e. childless) Glyphs bind to any data. The client can then Set and Get the value of the Property and determine its run-time Type. When doing a Set, the client must submit all the variables to a Multisetter component, which would ensure that BIO Multiple Set is used for group storage operations directed at the same BIO. The use of Multiset is imperative because validation rule interlock may occur otherwise.

<u>Default Glyphs</u>

The system can generate default Glyphs for aggregate data types, including BIOs and Bags. This functionality is available both at tool time to offer the analyst a starting point from which to fine-tune the presentation, and at run-time if a terminal Glyph binds to an aggregate data item.

The default Glyph structure for a BIO consists of a parent with Layout Policy = "Row" and one child Glyph for every Property within the BIO, in the order of appearance.

If the Property Type is Float, Integer, String, Text, Flag, Time, Currency, or a similar atomic type, the corresponding child Glyph has no children and no attributes. Rendering of the variable is governed by system-wide defaults and hard-coded default behavior within the Interactor.

If the Property Type is indicated as "IUnknown," the Property is of an aggregate type. The value of the property is in this case an IUnknown pointer. The code uses Get to obtain this pointer, and then queries the object to determine its type: If the object supports either the IBD interface or the IBag interface, the corresponding child Glyph renders it as a hyperlink (a.k.a. hotspot). The action of the hotspot is to create a new top-level Glyph below the current form to represent the linked BIO or Bag. (This behavior is known as the Story Browser.) If the object is neither BIO nor a Bag, the Glyph is left blank and rendering is up to the hard-coded Interactor implementation.

The default Glyph structure generated for a BIO also includes "OK" and "Cancel" buttons at the bottom of the form.

In the case of a Bag, the default Glyph consists of a parent with one child Glyph for every item in the Bag that is either a Property, a BIO, or a Bag. The parent has Layout Policy = "Tab" and Layout Style = "Picklist". The combination of "Tab" and "Picklist" cause child Glyphs to appear one at a time in the left hand side of the screen, with a selector listbox on the right hand side. Child Glyphs are constructed as appropriate for each item in the Bag in the same way as for BIOs. The listbox is populated with whatever the Interactor code would normally use as the label for the corresponding item. (In the absence of the Label attribute, either the internationalized Property name or, if not available, the value of the first Property inside a BIO is used as a label.) At the bottom of the form the default Glyph includes a "New" button that, when clicked, adds to the bag a blank copy of the object currently displayed.

Interactors for Web channels

Client-side Java applets and ActiveX

An important feature of late model browsers is the hosting of Web pages that may contain scripts, Dynamic HTML, ActiveX controls and Java applets. There is currently disagreement as to whether downloadable applets and ActiveX should be used freely or sparingly on -powered web sites. While applets and ActiveX controls can provide a richer and more attractive UI, downloads may be annoying to a user who is merely passing through the site, or to one who just wants to get simple pricing information in a hurry. In any event, however, at least some applet/ActiveX integration is required to display grid and tree controls, an occasional "rotating image" control, and the like.

The principal distinction between applets and ActiveX controls is that with applets Java bytecode is downloaded into the client, while for an ActiveX the binary is downloaded. As a result, the same applet can run on all computer platforms, but an ActiveX only runs on the platform for which it is compiled. Furthermore, applets are supported by both Netscape Navigator (NN) and Microsoft Internet Explorer (IE), whereas ActiveX can only run in IE. The applets' downside is that while both applets and ActiveX can expose methods to the script within the hosting Web page, only ActiveX can naturally throw events. Event throwing is needed, for example, to inform the script that the user expanded a branch in a tree control, so that the script can request from the server the data to populate the branch. (Within the architecture, it is generally not possible to download the entire tree control data into the client in advance).

Fortunately, it is now becoming possible to reach script functions from a Java applet after all, although the techniques are different in NN and IE. In Netscape, one must use LiveConnect, specifically the win.call() method. In IE, event firing from applets is supported starting with version 4.0; see Q178994 in Microsoft Knowledge Base and "Creating an applet using scripting" on the MSDN CD.

<u>Client-Server Communication</u>
<u>HTTP protocol issues</u>

Regardless of the client's use of applets for presentation purposes, its communication with the server is still contained within the HTTP protocol. In other words, we <u>do not</u> at this time plan to implement direct communication between the server and client-side applets or ActiveX via non-Web mechanisms such as DCOM. This is because the basic server access must be available to clients that cannot run applets or refuse to run them for security reasons. Furthermore, an off-the-shelf copy of Windows95 without an additional developer kit installed does not support DCOM objects communicating with other DCOM objects across the network (this feature is built into Windows98, however).

One fundamental limitation of HTTP is that any conversation is (1) initiated by the client; (2) results in the reloading of a Web page; (3) is closed as soon as the page is reloaded; and (4) offers no natural way to preserve state information through the reload. If state information has to be preserved, one must either (a) communicate it to the server and back, or (b) store it on the client as a cookie, or (c) store it as a script variable within a separate frame that is not being reloaded.

Cookies are character strings stored on the client's hard drive. Although some weaker browsers may not support cookies and some may reject cookies because of (unjustified) security concerns, we can probably assume the cookie feature to be available. (Cookie-less clients will experience difficulties in any event since the ASP server uses cookies for session identification.) A more serious problem is that the browser might restrict the number and size of cookies to protect the local hard drive: the "Netscape ONE" book mentions a limit of 20 cookies per originating domain.

The Client Side Framework

To work around the limitations of HTTP, client-server communication involving Web clients operates as follows. The browser screen is divided into several HTML Frames under the control of the Viewport Manager, designed to simultaneously present multiple documents (one document per frame) that may not be aware of one another. Each Document is generated and maintained by its own Composer, which is alone aware of the Document's structure, data binding, and behavior. A Document can contain scripts, Dynamic HTML, Java applets, and additional HTML frames (not controlled by the Viewport Manager).

A separate HTML Frame, called the transaction server Gateway, is reserved for system use. The transaction server frame is lightweight and fast to reload, perhaps empty of visual content or containing only a reference to a browser-cached logo image. The system also makes use of the master frameset file, which is never unloaded unless the user navigates out of the site. The master frameset contains script API code, referred to as Client Side Framework, or CSF, that can be reached by scripts running in any frame as top.foo(). For greater abstraction, however, Documents should use local wrappers, described in this section, that the framework inserts for this purpose into each frame.

All communications from a Document to the server are implemented through the API call:

```
OtsSubmitMessage(DocID,cmd)
```

The call causes the transaction server client-side framework to post a form that invokes a server side script OtsGateway.asp and ultimately reloads the transaction server Gateway frame with the return value of the script. The DocID parameter (a decimal integer) identifies the Document to the server so that the request cmd can be routed to the appropriate Composer. (Each Composer is initialized with a unique DocID, which it must imbed into all HTML pages it builds.)

The Composer executes any actions to fulfil the request generated by its Document, such as storing values of bound data objects. If the Composer also needs to change visual appearance of the Document, it submits to the Viewport Manager an Updategram script. The updategram can range from minor corrections using appropriate Dynamic HTML calls to a full reload by means of document.write(). System triggers may in some cases cause other Composers to submit Updategrams even though the request did not come from their Documents. The Viewport Manager assembles all Updategrams into a single script and passes the script back to OtsGateway.asp. The combined script is then loaded into the transaction server Gateway frame for execution inside the client browser.

Upon its arrival at the client browser, the script "signs in" by calling

```
top._OtsScriptSignin(SeqNo)
```

with the unique sequence number assigned to it by the transaction server server code. The number will be passed back to OtsGateway.asp with the next client request. Request sequencing helps detect lost packets, packets arriving out of order, and the use of the browser's "Back" button. Document Composers are thus isolated from the asynchronous nature of the underlying HTTP protocol: that is, any command a Composer receives is guaranteed to have been generated by the latest version of the Document this Composer had posted.

A Document's Composer can in principle write its output into a server-side file and direct the client browser to load this file by calling location.replace() from the updategram script. If the file is an HTML file (rather than ASP), the browser can in some cases cache it on the client hard drive and save a network transfer. Nonetheless, this practice is discouraged for two reasons. First, writing a server side file slows down the server, especially if multiple clients do so. Second, the Composer would not know when the file has reached the browser, and may continue to receive commands from the previous ("ghost") versions of its Document.

Browser Identification

IIS-defined SessionID is insufficient for transaction server session identification. For transaction server purposes, two browsers on the same computer represent separate sessions, both when one had been created by a script call from the other, and when the two were opened independently by the user. Any 'temporary cookie'-based identification scheme, including that implemented by IIS, would view the two browsers as separate only in the latter case.

We distinguish sessions by BrowserID, which is assigned at the time of first hit and which the browser maintains in a page variable. The page variable is lost if the user leaves the site and later navigates back, however, the session can still be identified because a request fired from the history stack, as any request, carries the BrowserID. BrowserID is also restored if the page is pulled out of the client cache, in that case by the page's Onload handler.

Client State

Any changes made through Dynamic HTML, as well as any as yet unsaved user keystrokes, may cause the client-side Document to diverge from its server-side source. Here "save" refers to the storage of the keystrokes in the HTML source file for the Document, as distinct from the storage of the entered data in bound variables. (A new value may in principle be already stored in the Context but not yet reflected in the HTML source for the Document, although such practice is discouraged.) The differences between the Document and its source are in danger of being wiped out if the page is reloaded ("loss of Client

State"). This happens in particular when the Viewport Manager must move or resize the Document's host frame, and when the user follows a link out of the site and later returns. It is recommended that the Document cooperate with its Composer to synchronize the server- and client-side image of the page every time a request is placed from one to the other. The Composer can update HTML source, for example, by adjusting default value attributes of HTML controls, or by inserting script code to expand and populate the branches of a tree control.

Nonetheless, the Viewport Manager must sometimes reload a Document whose state has changed since it last posted a request. Even if the Document were to immediately post its current state, the request might get stuck somewhere in the Internet and become outdated as the user keeps typing into the page. The user can also spontaneously navigate out of the site, in which case even the Viewport Manager receives no warning of the pending reload.

To preserve their "hot" data against sudden reload, Documents can submit it to the CSF for storage. This is accomplished with the API call

```
OtsSaveState(DocID, varName)
```

Here the second parameter is a string that evaluates to a fully qualified frame-scope variable name with an atomic type, for example "myForm.myField.value". The name, furthermore, must be in the plain dot-separated format (i.e., "myForm["myField"].value," although legal JavaScript, would be unacceptable). The framework evaluates the name in the context of the calling frame, and stores the value within the master frameset in an array keyed on both DocID and varName. If the master frameset itself is shuttled out of the browser, the data is preserved in a temporary cookie.

Once submitted to CSF through OtsSaveState, the variable is monitored for the lifetime of the Document. Specifically, CSF automatically fetches its value whenever the Document is shuttled out of the browser for any reason, and automatically restores it whenever the Document is shuttled back in. The CSF scripts would not crash if subsequent updates eliminate the variable from the Document, however, the item would remain in the master frame array. It is usually unnecessary for a Document to remove a variable from storage, however, an API call could easily be provided for the purpose if needed.

Since Documents often need to transmit to the server the same set of variables that they need to preserve across reloads, a special API call permits to reuse the state storage array for this purpose.

OtsStateToString(DocID, mask)collects the latest values of all variables previously registered with CSF via OtsSaveState, and packs them into a string in the 'key1=value1&key2=value2…" format similar to that submitted by an HTML form. The only difference from format is that the values are escaped rather than URL-encoded. Both Escape and URLEncode are conversions of a string into a form suitable for transmission via HTTP protocol, as part of a URL, or as a quoted string. Both replace special characters within the input string with the '%' character followed by the hex code of the character. The main difference between the two schemes is their treatment of the space character: Escaping encodes it as '%20' and URLEncoding uses '+'. An escaped string would be read correctly by a URLUnencoder, but not vice versa. The optional mask parameter is a regular expression which the caller can use to narrow down the set of variables included in the output.

It is up to the Document developer to choose how closely the source files on the server will track the ever changing situation within the client browser window. Some client activity, such as showing a rotating ad, is entirely presentational and its state need never be preserved. Other state information, such as text typed into an edit control, can perhaps be maintained on the client until stored in the transaction server workflow variables.

Miscellaneous notes

Netscape has been found to simply not call the OnUnload handler in cases when the frame hosting the Document is itself being destroyed at the time the Document is unloaded, or soon afterwards.

How the server deals with the browser's "Back" button is as yet undetermined. Javascript's window.open() apparently allows to run the entire session in a new window without any browser toolbar, however the history list would still be available to the user through Alt/Left-arrow. Alternatively, rumors have been circulating that it might be possible to programmatically disable or flush the browser's history list. Yet another option is to use sequence numbers to identify requests fired out of the history list.

The Web UI involves frequent dynamic changes to frames and the frame structure, causing, despite our best efforts, frequent reloads and screen flicker, especially with an overloaded server or slow network connection. The web site may not look as attractive as some Microsoft Web pages that seem to open up additional screens instantly. Microsoft achieves this elegance by downloading all data at once but initially hiding most of it using Dynamic HTML.

Server/Synchronizer Model

The Synchronizer enables all the objects in the Architecture to collaborate anonymously. All objects notify the synchronizer of significant events. The Synchronizer maintains a list of Workflows and Rules which have registered an interest in particular events. When an event occurs the Synchronizer invokes the interested BWs and Rules.

The Synchronizer establishes an anonymous subject-observer Pattern where every object in the system is a subject and the registered Processes and Rules are the observers.

The primary uses for the Synchronizer are:

- To listen for Peripheral notifications and spawn appropriate BWs.
- To provide preemptive Rule evaluation - objects can be blocked by Synchronizer rules they don't even know about.
- To handle asynchronous, unpredicted events - such as an operator terminating a Process or a server crash.

Related Objects

The synchronizer abstraction is comprised of several collaborating objects. The important ones are described below.

Events

Event Overview

Events are flyweight objects that are used throughout the transaction server. An event consists of a set of fields used to describe an incident that is of interested to the transaction server. Every event has standard fields such as category (where it originated) and ID (what it describes), in addition to a special info field. The Event info is an opaque, "black box" VARIANT field that can store anything. If the info field is of a certain format, the Synchronizer can use it; otherwise, the Synchronizer ignores the event info. Many events are used by various objects in the transaction server but never reach the Synchronizer at all.

Event info

If an Event's info field is a SAFEARRAY the Synchronizer might use the event to perform interesting work such as firing Workflows or queuing Steps. The Synchronizer uses the following procedure to determine whether to use the event info:

The SAFEARRAY must be one-dimensional with elements of type VARIANT.

The names may be interned BSTRs or raw BSTRs. Although interning the standard tokens is strongly recommended.

Synchronizer interpreted VARIANTs in the safe array must be of type BSTR with one of the following names:

- @session: a pointer to the controlling IUnknown of a valid contact channel session interface.
- Next element: VARIANT of type IUnknown containing the pointer.
- @property: a property to be preloaded into the context of whatever workflow is started
- Next element: Controlling unknown of the property.
- @CursoryID: the user ID of the user who is running this workflow.
- Next element: the user ID of the user (BSTR).
- @StationID: the station ID of the station at which this workflow will be running
- Next element: the station ID (BSTR)

Any other elements in the SAFEARRAY are ignored by the Synchronizer.

For an event to start an interactive workflow, the @session element - at a minimum - should be in the info-SAFEARRAY field. If it is not, the workflow will still be able to run, but it will not have a default session. Other entries are optional. If provided, the Synchronizer will preload the Context with variables, or it may route the incoming event to be dispatched in the session matching a station ID that the event specifies.

The framework includes a class, transaction serverEvent, which eases the task of creating the SAFEARRAY for the particular IEvent.

<u>Synchronizer and Events</u>

Workflows are started by the Synchronizer, usually in response to an Event received by a Peripheral (Web, Email, etc.). The Peripherals involved merely notify the Synchronizer of significant events; they may or may not know that a Workflow will run. The Event carries two kinds of information:

- Identification: matched by one or more Selectors; this includes the Event scope, instance and ID.
- Information: arbitrary information ignored when determining which Selector matches the Event, but used by the matching Selector's handler.

How Event Data is Used

A Typical Example

The Identification portion of the Event is described elsewhere in this document. Here, we describe how the Data portion is used when starting Workflows. First, let us provide a brief overview of how a typical Workflow is started: A customer hits the web site, IIS fires a short (usually one-liner) ASP file which invokes the Router, the Router prepares an Event (by creating an HTTPSession and packing an interface to the IC2Session into the Event's Data field) and sends the Event to the Synchronizer. The Synchronizer runs the Event down its list of Selectors, and the Selector that matches this Event has the SEL_PROCESS style. The SEL_PROCESS handler looks at the Event's Data field to get the interface to the web session, creates a Context for the Workflow, and starts the Workflow.

Meanwhile, while the BW runs, the Router is waiting for a response from the Synchronzer. The BW will either end, or it will perform a Render operation, waiting for a response from the channel on which the Render occurs. In either case, the Synchronizer returns and the Peripheral that sent the message continues accordingly.

Event Data Details

The data portion of the Event is a VARIANT. This data is used in different ways by different Selectors or event handlers. It can be a single data item, such as a string or pointer, or it can be a SAFEARRAY collection of data. In order for the data to be used properly, the Event sender and the Event handler must agree how to interpret it. The only restriction is that COM must be able to marshal the data. In almost all cases, the information in the Event will be the Context associated with the BW that fired the Event. This will be evident in Table 5 below, which shows how some of the Synchronizer's standard event handlers use the Event data field:

Table 5

SELECTORSTYLE	EVENT INFO VARIANT TYPE	USAGE
SEL_STEP	VT_UNKNOWN	The controlling Unknown for the Context in which the Step will be queued.
SEL_PROCESS	VT_UNKNOWN	The controlling Unknown for the IC2Session interface for the session in which this Workflow will run.
	VT_UNKNOWN	The Context of the BW that fired the event. The IC2Session interface, if any, can be derived from this Context.
	VT_ARRAY	A SAFEARRAY of name / value pairs, using syntax described above.
SEL_PROCESS_ ROUTE	VT_ARRAY	A SAFEARRAY of name / value pairs, using syntax described above.
SEL_ROUTE _WORKFLOW	VT_ARRAY	A SAFEARRAY of name / value pairs, using syntax described above.
SEL_PROCESS_ ROUTE_TERMINATE	VT_ARRAY	A SAFEARRAY of name / value pairs, using syntax described above.
SEL_RULE	VT_UNKNOWN	The controlling Unknown for the Context in which the Rule will be fired.
SEL_CALLBACK	VT_UNKNOWN	The controlling

		Unknown for the Context in which the callback will be invoked.
SEL_REENTER	VT_UNKNOWN	The controlling Unknown for the Context (same as the handle described below) of the session to be reentered.

Peripheral - Synchronizer Event Protocol

Here we describe the event protocols for carrying out various synchronizer features:

Starting a New Workflow in a New Session

The peripheral prepares an Event by setting a Scope, ID and Name, and creating a SAFEARRAY of VARIANTS to stuff into the Event info field. If this were happening on the Web channel, the Scope would be Web Contact Channel, the ID would be 2 (start new session) and the Name could be anything (as long as the Synchronizer has a selector matching this name). The SAFEARRAY would contain the controlling unknown of the channel object that would handle this session. It might also contain the controlling unknown of properties that the web channel wanted to have preloaded into the context of the workflow to be executed. When the channel sends the event to Synchronizer, it (that thread) blocks, waiting for the Synchronizer to respond.

The Synchronizer receives this event and it matches a selector whose style is SEL_PROCESS. That selector says to run the Foo workflow, so the Synchronizer creates a context, adds any variables specified in the Event info SAFEARRAY to the context, attaches the Event info Session to the Context, and starts the workflow.

Meanwhile, the thread in the channel is still waiting for a response.

The workflow begins running and reaches a point where it ends (unlikely), or it encounters a blocking Render action (more likely). So the workflow ends or suspends, which in either case returns a result code to the Synchronizer, which returns this result code to the channel. If the workflow is suspending upon a blocking Render (the usual case), it gives the channel a "handle" before it returns. The channel treats this handle as an opaque item and uses it later as described below.

The channel, upon returning from the Synchronizer call, checks whether the workflow posted any information to be rendered (it usually will). If so, it calls the Composer to render the information, and the user sees the initial screen of the workflow.

Reentering a Running (Suspended) Workflow

Now let us suppose the operator has finished with this screen and clicks Next or Proceed. The channel creates another event and sends it to the Synchronizer. The event Scope is Web Contact Channel, the ID is 3 (reenter session) and the Name is not needed. The channel packs into the event info field the handle that the workflow gave the channel when the workflow first called Render.

The Synchronizer receives the event and it matches a single common "reenter" selector. The selector style is REENTER, so the Synchronizer unpacks the event info field (which is the handle) into the Context of the workflow that should be reentered. The Synchronizer tosses this Context to the workflow and it resumes, picking up where it left off, immediately after the original call to Render.

Pushing an Asynchronous Screen Pop (Workflow) to an Existing Session

Sometimes a peripheral receives an event that must be routed to a user who is already running workflows on the transaction server. This is unlike the above cases because there, the peripheral that reported the event also provided the session used to handle the event. Here, the peripheral reporting the event expects some other session to take care of it. For example, imagine a CTI system where an agent uses his web browser to log into transaction server and start working, and while he is working calls come into the CTI switch and the CTI Peripheral needs to tell the Synchronizer to push a screen pop out to a particular agent's browser.

As usual, the Peripheral starts by preparing an Event. This Event will have a Scope of Telephony Contact Channel, an ID of 5 (delegate handler), and any Name. The CTI Peripheral will use a SAFEARRAY for the Event info field. For a new call delivery, this SAFEARRAY will contain a CallObject BIO (describing the incoming call) to be preloaded into the Context, a Station ID describing the station to which the event handler should be routed (currently, the Station ID is the fully qualified phone number of the agent to whom the call was routed), and the controlling unknown of a Session the workflow can use to carry out activity on the call (forward, hold, etc.). Observe that this is not the session on which the workflow will run (the Synchronizer will choose that session), but it may be used by the workflow to carry out CTI activity pertaining to this call. Additionally, the event's info field will also have a CallContact BIO which describes the current call as well as a CallContactSegment BIO which documents the current contact segment. The CTI Peripheral sends this Event to the Synchronizer and it (the CTI Peripheral's sending thread) waits for a response. The threading model of the CTI Peripheral is designed so that none of these waiting threads block other call activity from happening.

The Synchronizer receives the Event and finds a matching Selector. The Selector style is SEL_PROCESS_ROUTE. This causes the Synchronizer to grab the Station ID from the Event info SAFEARRAY and store the Selector's handler (workflow) name in the Synchronizer's information record pertaining to this station. The Synchronizer then creates a new Context and notifies the Listener object living in the browser of the agent at the given station (the listener's address was provided to the Synchronizer when the agent logged in). This notification includes the controlling unknown of the new Context, which the Listener receives and treats as an opaque handle. On the client, the Listener opens a new browser window on the agent's screen and returns to the Synchronizer. The Synchronizer thread then returns to the CTI Peripheral, which has now completed its role in processing this event.

Meanwhile, the new browser prepares another Event and sends it to the Synchronizer. The new Event has Scope Web Contact Channel, ID 6 (delegate browser), and the Context handle packed into the Event info field. The Synchronizer receives this event and it matches another Selector whose style is SEL_PROCESS_PUSH. This Selector's handler grabs the Context handle and workflow name that it remembered from the previous Peripheral event, attaches it to the session at the given station ID, and starts the Workflow. When the workflow renders, it creates a screen in the newly created browser, causing a pop-up on the Agent's screen.

<u>Detailed Synopsis</u>

Call arrives

CTI sends event to Synchronizer.

Event Scope: Telephony Contact Channel

Event Name: NULL (or anything)

Event ID: 5

Event info: SAFEARRAY of VARIANT containing:

Station ID

IC2Session for call handling

CallObject BIO

CallContact PBIO

ContactSegment PBIO

Synchronizer matches event to a Selector:

Selector Scope: Telephony Contact Channel

Selector Name: NULL (matching event)

Selector ID: 5

Selector Style: SEL_PROCESS_ROUTE

Selector Handler: Workflow name

Synchronizer goes to the lineage corresponding to the station ID, gets the Listener interface.

Synchronizer stores the workflow name (selector handler) with this lineage in the session list.

Synchronizer creates a Context for the new Workflow, builds a VARIANT handle from the Context.

Synchronizer adds this Context to the lineage using IWorkflowRegistry.

Synchronizer sends a message to the Listener, including the handle.

Listener creates a new browser, gives it the handle.

New browser starts up, creates a new Session.

New browser sends event to Synchronizer:

Event Scope: Web Contact Channel

Event Name: NULL (or anything)

Event ID: 6

Event info: SAFEARRAY of VARIANT containing:

VARIANT handle (provided by Synchronizer)

IC2Session (of new web session)

Synchronizer matches event to Selector:

Selector Scope: Web Contact Channel

Selector Name: NULL (matching event)

Selector ID: 6

Selector Style: SEL_PROCESS_PUSH

Selector Handler: NULL (not used)

Synchronizer finds the lineage associated with the incoming event's VARIANT handle.

Synchronizer starts the workflow whose name was stored in step 5, using the web IC2Session interface provided by the incoming event.

Business Rules

Business Rule Overview

Business Rules are conditionals. They check a condition and return a status to their caller. One can think of them as read-only, or passive Operations. A Rule queries information in a BIO, optionally performs some calculation, and returns information to the client. Business Rules determine application behavior. A Business Rule is a condition used to determine the behavior of an activity within the system. Business Rules determine (indirectly, but ultimately) to which Step a BW will jump next; they validate input accepted from Peripheral sources; they determine whether a given Operation is to be taken, etc. If no Business Rules are present to determine which Step is the next Step, the BW will jump to its next "required" step. If all its required steps are complete, the BW will terminate and return to the next nesting level.

Business Rules consist of three parts:

Qualifiers are used to determine whether a Business Rule is relevant in a particular combination of BW Step and Context. These are optimized for speed and acted on as a group for fast evaluation. An example of a Qualifier would be a permission setting. In the case of a "default next step" Business Rule, the Qualifier simply identifies the step in the BW to which the rule applies.

Success Conditions are used to determine whether a qualified Business Rule should permit the triggering Operation to continue. If the Success Conditions end up as TRUE, the Business Rule will return a success code. If not, a meaningful failure code and, optionally, a failure message will be returned.

Context Effects are settings the Business Rule alters in the Context on success and/or failure. One typical Context Effect would be to change the next Step for the BW.

When triggered by an Operation, Business Rules operate as follows:

A BW Step fires an Operation.

Before the Operation is actually executed, it checks for Business Rules with Qualifiers met in this particular combination of Operation and Context. This check is performed as a part of the private implementation of the Operation; thus is hidden from the Step.

Any qualified Business Rules test their Success Conditions and report status codes.

If no qualified Business Rule fails, the Operation proceeds.

If a Business Rule fails, the Operation fails and the BW can take operation based on the return status code.

Business Rules are reusable. Business Rules may be reused across BWs, and therefore a given Business Rule is never to be considered to belong to a single BW. Business Rules exist independently within the Process Manager. A Process uses Rules, but it does not own them.

In case the same Business Rule pertains to more than one of the Operations within a given BW, it will be possible to refer to the response from the first consultation of the business rule. This is optional, as it may be desirable to reevaluate the business rule. This information would be stored as part of the Rule's private data, thus used by the Rule automatically and transparently to the Step or Operation using it. In other words, some Rules may cache their results.

There are two types of business rules: Simple and Normal. Simple business rules are those that are suitable to be cached in the interface logic on a UI. This facilitates the ability for users to have conversations in the UI while offline, but requires that the rules make their decisions based only on that portion of the Context which can easily be cached within the UI. Complex business rules are those that require contact with the server. In order to maintain complete delivery channel independence, the same rule may be realized as both simple (in the UI) and normal (in the server). For example, suppose an application must verify that a customer's address has a valid ZIP code. A web GUI might use a simple Rule to check that the ZIP code has 5 (or 9) digits, and BW might use a normal Rule to check again. This way, the web GUI could catch this simple error before starting a transaction on the server, but the same BW would still function when deployed across another delivery channel, such as a voice mail interface, that lacked the ability to check the ZIP code.

Business Rules are also used by BIOs in the Cache. See the section on BIOs below for more information.

String Management

The framework, being a meta-data driven system, uses strings extensively. Object class names and instance names, and property names represent some of the key areas in which the architecture depends on efficient string management.

The Scope of the String Management Problem

Imagine 1,000 agents using the server. Picture them running a typical Workflow which has 50 variables in the Context, 30 of which are BIOs which each have 20 properties. That is a total of 620 properties multiplied by 1,000 agents which is 620,000 properties. Each has a class name, instance name and property name, which are strings of approximately 10 characters each, for 30 characters times 620,000 properties is 18.6 MB of characters (the fact that this is exactly 100 times the speed of light measured in miles per second is merely coincidence). But remember that this is all Unicode, which doubles the data size to 37.2 MB of string data. And this does not even include the values of the properties in question!

The Magnitude of the String Management Solution

If we could introduce a singleton string manager object we could drop the storage requirements by a factor of 1,000 - taking our 37.2 MB of string data down to 18.6 KB. And this would further be cut in half to 9.3 KB if the string manager stored strings as multi-byte rather than Unicode. In addition, since the objects themselves would maintain only IDs for the strings, they could perform string comparisons in a single instruction rather than using real string comparison routines.

In summary, in this example the string manager provides a 2,000 fold decrease in string storage requirements, faster string manipulation, and significantly reduced marginal resource cost for each additional user who logs into the system. What this boils down to is dramatically improved system scalability.

How String Management Works

The transaction server has a singleton object, the Translator, which maintains a hash table of all the strings being used in the system. The hash table stores the strings in a compressed format and provides lookups in near constant time, regardless of the size of the table.

Every object that uses strings either receives them already interned, or it interns them. The intern process returns to the object an ID for the string. The object hangs on to the ID and destroys/frees the original string it had created. The mapping from IDs to strings is a "bijection," which means objects can compare string IDs (rather than strings) for equality. If the IDs match, the strings match. If the IDs do not match, the strings are different.

Whenever the object needs the actual string, it sends the Translator a getString message and receives a copy of the string. The Translator will provide the string in any format the client desires - currently either multi-byte or Unicode. This returned string is only a copy and can be freed at any time, whenever the client is done with it. At first glance one might think that copying the strings in this manner would represent a performance cost. But there is no such performance degradation - indeed, it is a performance improvement - primarily for three reasons:

The object that owned the string would have to copy it anyway for its client, even in the absence of the Translator. This is because objects in a distributed object system don't in general share the same address space.

Converting the strings from the Translator's internal format to multi-byte or Unicode is not much more expensive than raw duplication, and we must duplicate them anyway.

The massive time and space savings realized from using interned string IDs more than counters the marginal cost of duplication needed when actual strings are needed.

Message (Phrase) Translation

The Transaction Server (transaction server) provides a uniform mechanism for translating application messages (phrases) to the language corresponding to the locale specified in each Workflow's environment. The translation mechanism does only that - translate phrases. Lookup resolution processing (for properties with domain value matrices) is a completely different topic handled by different objects (see next section).

When a business analyst builds an application, all phrases that may appear to the operator are specified by message ID or by message name. At runtime, the message IDs or names are mapped to phrases by a dedicated singleton object in the transaction server (the Translator). This translation may run in the opposite direction - sometimes a phrase may be provided by the user and must be reverse translated into the appropriate message ID (for example, in the case of a property with a domain matrix).

The Workshop enables the business analyst to define new message phrases and assign message IDs to them, and to update, insert and override translations for any message phrase.

How Messages are Identified

The subtleties of language translations make it impossible to store messages in a simple one-to-one phrase translation lookup. Often a single word in language A covers multiple concepts that are identified by different words in language B, and the reverse of this happens, too. Because of this, message IDs and phrases must be chosen carefully and in some cases the same phrase will have multiple different message IDs, depending on where/how it is used. When translated to a different language, these identical phrases might be different. For example, in English one talks of opening a file, opening an application and opening a socket - we can use the word "open" for three different concepts (reading, invoking and connecting). Another language, say Swahili, might use different words for these concepts.

Previously, for simplicity, we have mentioned language IDs. However, language is only one of many possible dimensions that can be used to resolve a message ID into a phrase. We might want to use different translations for different customers, tenants, or other permutations or states of business data. In general, we have multiple orthogonal dimensions upon which the resolution of a message ID to a phrase relies.

Phrase translation dimensions are stored in table s_qualmeasures, which is used to store all sorts of standard naming information.

How Messages are Selected

When a client object asks the Translator to provide a phrase, the client provides a message ID or name. The Translator finds the best matching phrase. The translator guarantees to always provide some kind of phrase in return, even if it must return the empty phrase.

The translator maintains a cache table of phrases that it reads from the database when it initializes. In the database, many different phrases may be provided for a given message and scope ID, as long as they have different language IDs or precedence (precedence is used to override OTB messages without modifying them). The client is not required to specify the language ID (though it can); normally, the Translator chooses a language appropriate to the client's environment.

It is possible to provide many different phrases for a single message ID, scope ID and language ID, though only one will be used a runtime. When this happens, the phrases are assigned a precedence and the Translator chooses the phrase with the highest precedence. This enables customers to override the standard application messages without modifying the original phrase meta-data.

How the Translator Defines the "Best Matching" Phrase

As mentioned above, the client provides the Translator a phrase ID. Here's what the Translator does:

If the client did not provide a language, assume the default server language. Set the language ID.

Find all phrases that match the specified phrase ID and language ID.

If at least one phrase is found, choose the one with the highest precedence.

If none are found, ignore the language ID and search for a phrase matching the message ID for the server default language.

If at least one phrase is found, choose the one with the highest precedence.

If none are found, return the empty phrase.

Translation - Internationalization

The appropriate phrase may depend on locale, language, tenant and other conditions known only at runtime and which may or may not be specific to a given client or use server defaults.

Consumer objects who want application phrases will invoke methods on the Translator object to obtain the phrases. Consumers will pass in a phrase name or phrase ID, optional translation dimension information (referred to above as "language ID" for simplicity, described in detail below), and will receive back from the Translator the appropriate phrase (in the form of a BSTR or an interned string ID).

Phrase Translation Dimensions

Phrase translation dimensions describe conditions for selecting the appropriate phrase for a given phrase name or ID. The simplest and most obvious way to do this is using a one-dimensional system of "language". For example, phrase 17 maps to "My name is" in English, but maps to "Je m'appelle" in French. However, this one-dimensional scheme may be insufficient for complex business applications. For example, what if the application wants to have formal and informal translations? The English version of the above phrase might not change, but the French version might change to, "Mon nom est". Further, what if different translations are needed for different tenants using the same application under a hosting environment (say, Wells Fargo and B of A sharing a call center for hosting customer service). Now we have at least three dimensions for translation (language, formality, tenant).

Instead of building a system with a fixed set of numbered and predefined dimensions, the framework enables business analysts to design phrase translation dimensions with no arbitrary limits to the names or number of dimensions.

How Translation Dimensions are Implemented

Since the number and names of dimensions are arbitrary, and since the order of dimension N-tuples is not relevant (English, Formal is the same as Formal, English), finding the matching dimension conditions at runtime could be prohibitively expensive. To avoid this problem, the Translator uses dimension Handles, which are 4-byte integers that represent a particular, pre-validated, dimension N-tuple.

Each client object invokes a method on the Translator to resolve its dimension N-tuple once at runtime, to get a dimension handle. It then uses this handle in subsequent Translation calls. A given dimension N-tuple will always have the same dimension handle for the up-time of the transaction server, but it might be different the next time the transaction server is booted. This is why clients must resolve the dimension N-tuple to a dimension handle at least once at runtime; it cannot store the handle to persistent storage and use it later with a different instance of the transaction server.

Finally, phrase translation dimensions are entirely optional. Clients may resolve phrase IDs or names to phrases without using translation dimensions. When clients do this, they get messages translated in accordance with the server default policy, which is set in meta-data.

Phrase Translation Implementation

Internally, the Translator uses a series of cross-referenced tables to maximize the speed and minimize the storage requirements of translation.

Phrase Tables

After all, phrase names and IDs have to be cross-referenced to actual phrases. The Translator interns all phrase strings and uses a phrase table to store the cross-references. A phrase table is a table consisting of two 4-byte integer columns. It is implemented as an array of structures, where each structure has two 4-byte integer fields. The first column of the phrase table is an index (either a phrase ID or the interned string ID of a phrase name), the second column is the interned string ID of the phrase string. The phrase table is built by the Translator upon startup, does not change at runtime, and is sorted along the first column.

Every translation dimension handle will have two corresponding phrase tables - one for phrase ID / phrase, the other for phrase Name / phrase.

Translation Dimension Table

As described above, for reasons of efficiency, the Translator does not accept translation dimension N-tuples in their raw form for translating phrases. Instead, it requires the caller to provide a phrase translation dimension handle. The caller obtains this handle by resolving a raw N-tuple with the Translator. The Translator uses a Translation Dimension Table to map N-tuples to handles. Every row in the table represents a phrase translation dimension N-tuple.

The translation dimension table consists of four columns:

- An integer indicating how many strings are in the array (see below).
- A pointer to an array of strings. The strings in the array are the names of the dimensions that comprise the N-tuple.
- A pointer to the ID-based phrase table for this translation dimension.
- A pointer to the name-based phrase table for this translation dimension.

As an example, the above dimension example would be represented as (3, [English, Formal, Wells Fargo], [ptr to ID table], [ptr to name table]) in the translation dimension table.

When a client resolves an N-tuple to a handle, the client gives the Translator an N-tuple and the Translator returns the corresponding handle. The handle is the index of the row in the translation dimension table that corresponds to the client's N-tuple (though the client treats it as an opaque data type). This index is zero-based, and zero is reserved in the translation dimension table for the server default translation dimension N-tuple.

How the Translator Resolves a Phrase ID to a Phrase

The caller provides a phrase ID and a translation dimension handle. The Translator uses the dimension handle as an index into the translation dimension table to find instantly the pointer to the appropriate phrase table. The Translator then finds the row in the phrase table corresponding to the caller-provided phrase ID (or name) and gets the interned ID of the phrase string. The Translator returns this string to the caller.

Lookup Resolution Processing

Overview

There are two categories of lookup resolution processing. The first is when we must select one element from a discrete, predefined set of values. For example, a "status" selector might be "open," "closed" or "pending". This is called a static lookup resolution. The second is when we must select one element from a dynamic list of data. For example, a selector that enables the operator to choose a customer from a dropdown list. This is called a dynamic lookup resolution.

In either case, they're called lookup resolutions because it would be (for many reasons) bad to store the selected string. At best, it is inefficient. At worst, it is ambiguous - more than one customer might be named "John Smith". Additionally, there may be translation issues. If a French user selects "ouvert" and an American selects "open," the status is the same. In all these cases, the solution is to store a unique identifier that represents the selected string. Every item in the selector must have a different identifier regardless of whether its string representation is unique.

Functional Requirements

Depending on the context of the selector, the same string may use different identifiers. For example, "open" (or "ouvert" or "abierto") regarding the status of a case might be 5, but "open" regarding the action to perform on an attached document might be 17. Of course, in a different language these might be two different words. But in general the words are irrelevant; the workflow and data objects care only about the status, which is expressed as a unique identifier.

Here it is important to provide great flexibility in the choice of identifiers. Message IDs are not sufficient, as many lookups (usually but not always dynamic ones) consist of proper names, thus will not be translated. Giving each selector linearly increasing numbering starting at 1 would be inefficient, as it would require some kind of numeric translation to the numbers appropriate to the domain of the selector. And since the domain space numbering would rarely be contiguous, such a translation would most likely consist of a table (rather than a function).

The Factory Control

Overview

The Factory Control is responsible for two tasks:

- CoCreating all transaction server factory objects and holding references to them until IFactoryControl shutdown() is called.
- Pre-loading transaction server factory caches.

Keeping References to All transaction server Factories

All transaction server factories (i.e. any object that implements Itransaction serverFactory) are FT/FTM singletons that keep various caching data needed to quickly build their respective objects. If the reference count for such a factory goes to zero, it will release all caching information as it deconstructs. This is undesirable since many more clients may make further requests after the "last" release and then the factory must be re-created and must re-stock its cache.

To prevent premature release of transaction server factories, the Factory Control CoCreates all transaction server factories and keeps a reference to each until IFactoryControl shutdown() is called. IFactoryControl shutdown(), calls shutdown() for each transaction server factory. This causes subsequent calls to transaction server factory methods to fail. Note that clients can still CoCreate these factories but their method calls will fail with transaction server_E_NOT_FOUND.

Pre-loading transaction server Factory Caches

Since all transaction server factories use caching to optimize creation speed, it is desirable to pre-load these caches in certain situations (e.g. making the initial start-up screen appear faster, etc.). These situations will vary between different installations so configuring factory cache pre-loading is essential. The Factory Control supports configurations via tables in the Business Model Repository (see the Business Model Repository section below for more information).

There are two cases for reading configuration information from the database:

- One Transaction Servers connected to one Business Model Repository.
- Many Transaction Servers connected to one Business Model Repository.

The first case is the usual case (especially at a customer site), and a Server administrator can use these various configurations to experiment with different pre-loading schemes. For the latter case, having many configurations in a database is the only way to enable different Transaction Servers to pre-load differently.

Default Configuration

Each transaction server has a default configuration that is specified via a registry setting. Upon initialization, the Factory Control uses this registry setting to load the transaction server's configuration for pre-load. This registry setting can be administered via the transaction server console application (i.e. transaction server console writes the default configuration value into the registry and transaction server Factory Control reads it on bring up).

Specifying What Gets Pre-loaded

To pre-load transaction server factory caches, Factory Control parses through its configuration and asks each mentioned factory to create the specified classes (i.e. all Factory Control needs is a list of {factory, class name} tuples). These requests cause the factories to fill their caches so that subsequent create() calls for the same classes will be very fast. See the Business Model Repository section below for more information.

Extensibility

The Factory Control is designed to support various actions. The only action we currently use is pre-load. but we are free to add more actions in the future as needed. See the Business Model Repository section below for more information on how the concept of action is used in the database tables.

Business Model Repository

These tables store information needed by the Factory Control. Typically, the Factory Control uses this information to pre-load classes. The tables are defined in Table 6 below.

Table 6

Table Name	Purpose
m_factoryctl	Each row contains the factory, class and action that the Factory Control should initiate. The ordering controls the order in which Factory Control performs its actions.
m_factoryctlcfg	Contains the list of configurations.
m_factory	Contains an entry for each transaction server factory
m_factoryctlact	Contains the list of actions that Factory Control can initiate.

Peripheral Model

Email

Introduction

Microsoft has recognized that developers need an object library providing greater functionality than what was available in the Active Messaging 1.1 library, which shipped with Microsoft Exchange 5.0. Developers need objects that support capabilities beyond simple Messaging and into the areas of

Calendaring, Collaboration, and Workflow. Such capabilities can simplify the development of heavy-duty resource-scheduling applications requiring information to be displayed through a calendar.

In response, Microsoft has replaced its Active Messaging library with Collaboration Data Objects (CDO). These CDO's are available in a library known as CDO 1.2, which ships with Exchange Server 5.5 and Microsoft Outlook 98.

CDO 1.2 is a scripting-object library that developers can use to design applications on both the client and server-side. It's an in-process self registered COM server which is language independent and can be used with many programming languages, e. g. Microsoft Visual Basic Scripting Edition (VBScript), JavaScript and many more, sometimes even C++.

CDO 1.2 can be used to build solutions, which are running on client- or server-side. It supports multiple concurrent sessions (according to Microsoft ~1,000 users per server). You can build client applications or ActiveX run-time controls, which are logged-on to Microsoft Exchange Server with another account that is currently logged-on with Microsoft Outlook 97/98. For example, the Microsoft developers have used the CDO 1.2 HTML Rendering Library to build the Microsoft Outlook Web Access.

You also have the choice to logon authenticated or anonymously to a Microsoft Exchange Server. The reason for an anonymous logon could be, to show all items of a public folder at a Web site without prompting the user for a logon.

With the release of Microsoft Exchange Server 4.0 at the beginning of 1996 Microsoft has introduced the OLE Messaging library in the version 1.0 to give developers the ability writing applications on top of Microsoft Exchange.

With the release of Microsoft Exchange Server 5.0 in Q1 of 1997 Microsoft has renamed the OLE Messaging library to Active Messaging library and included new functionality such as using

Active Server Pages technology to access Exchange information (also known as Outlook Web Access) and updated it to version 1.1.

With the release of Microsoft Exchange Server 5.5 in Q4 1997 Microsoft has again renamed this technology from Active Messaging to Collaboration Data Objects (CDO).

These CDO's are available in a library known as CDO 1.2 (included in Microsoft Exchange Server 5.5) and CDO 1.21 (included in Microsoft Outlook 98 and Microsoft Exchange Server 5.5 Service Pack 1), which replaces the Active Messaging object library version 1.1.

On the Microsoft Exchange Conference '98 in Boston USA, Microsoft has announced their plans about future versions of CDO. They are currently developing two different versions of CDO:

The first version will be CDO 2.0, the follow-up version of CDONTS. CDO 2.0 will be included in Windows NT 5.0 Server and Workstation. CDO 2.0 is already available as a beta with the Microsoft Windows NT 5.0 Beta 2 for all members of the Windows NT 5.0 beta program and all MSDN subscribers.

CDO 2.0 will provide basic messaging services, such as send e-mail, post discussions/news, support for the SMTP and NNTP protocols and inbound protocol events and agents. However, it does not provide access to mailboxes.

The second version will be CDO 3.0, the follow-up version of CDO 1.2. CDO 3.0 will be included in the next release of Microsoft Exchange Server. According to Microsoft, a first beta of CDO 3.0 will not be available before the mid of 1999 and the final release will come with the next version of Microsoft Exchange Server (code name 'Platinum'), 90 days after the release of Microsoft Windows NT 5.0.

CDO 3.0 will have a different object model than CDO 1.2 and will include extensions to the OLE DB and ADO programming model. It will be a superset of CDO 2.0 and will enable new functions, such as voicemail, fax and pager support. Also included will be enhanced scheduling, task and contact management and support for MIME headers (including S/MIME) and MHTML (HTML message bodies). Also advanced features for building collaborative application development (e. g. voting buttons and message flags) will be included. Last but not least, a future release of Microsoft Outlook will also support CDO 3.0.

CDO Session support in transaction server.

The transaction server uses Collaboration Data Object (CDO) library to provide Mail API (MAPI) support MAPI objects representation in CDO is shown in Fig. 5

Fig. 5 shows MAPI objects represented in the CDO object model. The top object on CDO is a Session object 510. Session object 510 contains session-wide settings and options. It also contains properties that return top-level objects such as CurrentUser. It's the only object that can be created directly from application.

After creation of a new Session object 510, the Logon method should be invoked to initiate a session with MAPI. No other activities with CDO are permitted prior to a successful logon, even getting any other method or property of the Session object. An attempt to access any programming element prior to a successful Logon results in a CdoE_NOT_INITIALIZED error return.

Transaction server E-mail system modules.

The main email module is Email Manager, which is system-wide MTS-hosted FT/FTM singleton. It provides MSMQ connection/notifications, messages dispatching and startup/shutdown sequences. It also provides coordination between e-mail peripheral system and peripheral script factory. Email Manager exposes itself to the system through MTS-hosted proxies, however it supports transaction serverGlobalSingleton functionality to reduce thread switching for subsequent calls.

Every message is handled by another FT/FTM object EmailSession. This object combines email-client functionality similar to the standard email client like Reply, Forward, CreateNew etc as well as IC2Session interface. The instance of EmailSession object wraps every message proceeding.

Email Manager is notified about incoming message events by MSMQ notification mechanism. To post the information about mail events transaction server uses Exchange Event Agent. This Exchange Agent module is implementation of the Microsoft Exchange Custom Agent. It supports IExchangeEventHandler interface so Exchange server can be configured to notify this object about events. For the information about how to configure the Exchange Event Agent see the installation guide.

<u>Transaction server E-mail thread pooling.</u>

To provide efficient messages processing Email Manager supports thread pooling. Upon initialization, Email Manager creates fixed number of threads. This number is specified in registry and is set by transaction serverConsole program, section Server Configuration/Email Throttling. The idea is that significant part of email-related peripheral and workflow spend time waiting for I/O operations like mail/database access. This means system is able to proceed few messages in different threads concurrently. The only limitation is that when number of such threads grows the thread-switching redundancy should be considered. Therefore number of concurrently proceeded messages is individual per computer configuration and is determined by factors like number of CPUs, network traffic and should be set during tuning procedure.

Another issue is pooled thread COM model. In the first email system implementation every thread from the pool initialized itself as belonging to Multi-Threaded Apartment. This was very inefficient because FT/FTM (free-threaded with free-threaded marshaller) component cannot create STA object on their thread. System either reuses main STA thread for such objects or creates (once) this STA thread and then reuses it. This force all e-mail related STA objects (like bags) to be created on the same thread and the same stack which can crash the program. Moreover, if the program even didn't crash, all STA objects created on the same thread can be used sequentially only. This serializes the entire system loosing all multi-thread benefits and puts an additional overhead because of the thread switching.

The only solution possible is to proceed every incoming message (peripheral and workflow) it its own dedicated apartment. This is the only way system can take advantages of usage the multithreading and avoid unnecessary thread switching because of mixed object with various COM models.

Transaction server Email Collaboration model.

The E-mail system object collaboration diagram is shown in Fig. 6.

Upon initialization Email Manager creates thread pool (stage 605). The pool size is then defined in registry and set by transaction server Console application (stage 610). To change the pool size, the transaction server has to be restarted.

Each thread belonging to the pool initializes itself as Apartment. To follow the apartment thread rules this thread has to pump Windows message loop on idle. This thread differs from CcomAutoThreadModule insofar as:

- It doesn't allow creating more than one instance of EmailSession object;
- It is getting the request to proceed not by custom message but by special event set by MSMQ notification object. This provides Operation System-based loading balance regardless of the pool size;
- It doesn't return pointer to the EmailSession object created to the MSMQ notification object. Instead, this

thread fully controls the entire lifetime of the EmailSession object created.

After initialization the COM, pool thread starts pumping Windows message loop waiting for the synchronization event raised by MSMQ notification object (stage 665). By obtaining the event it checks global data pointer for transmitted data and sets another event indicating "data is accepted" which lets the MSMQ notification object to restore the notification mechanism.

Email Manager's main thread also creates special thread, which provides MSMQ lookup (stage 615). The transaction server MSMQ can be found by queue GUID. Conceptually, the transaction server email system can work against multiple Exchange servers, which means one MS queue per server. Email Manager maintains list of these queues. Per each queue there is corresponding notification object created.

Email Manager lookup thread performs lookup every 30 seconds looking for queues with specific attributes like GUID, name, label, etc.

For each queue found, the thread creates a MSMQEvent object that provides Connection Point interface (stage 625). Then it advises Arrived and ArrivedError methods to MSMQEvent object. This provides MSMQ notification event to notify Email Manager of the message posted to the MSMQ (stage 630).

The MSMQ mechanism provides high fault-tolerance to the entire system. When the Exchange server receives a new message transaction server shouldn't be activated. MSMQ will guarantee posting message's information into MSMQ. When the transaction server email system is started, it will be notified immediately if there are messages waiting (stage 635).

MSMQ notification should be restored manually. Upon notification, the connection point procedure should call RestoreNotification method on MSMQEvent object (stage 640). This also means the system should work in configuration many-to-many (many transaction server email systems against many Exchange servers).

For each MAMQ notification, Email Manager extracts email message information from the queue and set "Data ready" event. Then it waits for "Data ready" event which will be set by the thread that accepted "data ready" event. Such event-driven mechanism provides first available thread will peek up the message.

For each "data ready" event received, pool thread co-creates EmailSession object which will be dedicated to the message received (stage 645). This object implements IC2Session interface as well as IemailSession, which provides basic mail actions like Reply, Forward, Delete.

EmailSession object performs peripheral actions (stage 650). This process is further described below.

Depending on the peripheral script execution results, email message might be sent to the Synchronizer (stage 655). This suggests that e-mail system delegated responsibility for the email message to the workflow but still can be used for rendering. At that point, email system thread and session become contact channel objects.

The Synchronizer can render message on the E-mail contact channel (stage 660). If email channel is the default channel for the workflow, the EmailSession object will be reused for rendering. If IC2Session interface wasn't provided to the workflow context or it belongs to another channel, EmailSession object will be co-created for rendering purposes. The rendering mechanism is very similar to the Peripheral script processing: the difference is that context "Message" BIO is used instead peripheral script. Please refer below to Email Peripheral design and implementation.

Upon workflow completion, EmailSession object released and pool thread restores messages loop pumping waiting for "data ready" event (stage 665).

Transaction server Email Peripheral design and implementation.

Eventually Peripheral scripts will be re-implemented to be a JScript or VBScript. For today, the current implementation is that script is just a linear sequence of text lines containing EmailSession object methods and properties. Despite table that defines script can be use for representation of any linear text line sequence, the only module, which uses these tables is the email system (Manager+EmailSession).

The m_selector table contains selectors for Email Peripheral. Selectors, which belong to the 'email' domain, as specified in mx_sel_seldomain table, will be loaded by EventHandlerLoader module per EmailManager request. In this case, k_handler points to the primary key k_handler in m_handler table specifying the Peripheral script (handler) name. These selectors are the same as standard Synchronizer selectors except the following:

Email selectors have either style STOP or style CONTINUE;

Email selectors contain peripheral handler index in k_handle column of m_selector table.

Every peripheral handler corresponds to the particular script that contains in m_handlerinfo table (cross-linked with m_handler table by mx_handler_hinfo). The sample of m_handler table is shown in Table 7.

Table 7

k_handler_info	s_handler_info
-14	::> from the subject line. It is used for tracking purposes.
-10	@@Reply
-12	@ConversationID
-11	@Insert
0	Not_Specified
-13	Thank you for contacting our Service organization
10	This is simple

The Peripheral script is defined by mx_handler_hinfo table which provided normalization for many-to-many links between m_handler and m_handlerinfo tables. As a result, the script would take the form:

```
        @@Reply
        @Insert
        @ConversationID
        @Insert
        "Thank you for contacting our organization, blah-
    blah..."
        @Append
        "Looking forward to here from you..."
```

The meaning of each entry in the script is described below.

The Sequential diagram of Email Peripheral model is shown in Fig. 7.

First, the Email Manager loads the Peripheral Selectors list using Synchronizer factory/loader (stage 705). After EmailSession object created, Email Manager calls it to proceed first selector with current message (stage 710). EmailSession checks whether selector matches the message's event (scope, id, instance name). If selector matches, the corresponding handler will be processed (stage 715). EmailManager is asked for handler (script), so it calls EHFactory (stage 720). EHFactory checks whether script is already cached and either returns the script (as an active bag) or calls EventHandlerLoader to load the script (stage 725) EventHandlerLoader loads the specified script from m_handlerinfo using mx_handler_hinfo (stage 730). The loaded script is an Active Bag object. Each of its entries is a SAFEARRAY of type VT_BSTR. The entry name is defined by first found text line which starts with "@@" prefix. Until subsequent entry starting with @@ is not found, the current active bag entry is filled with script lines. Then new active bag entry is created and filled with scripts such as the ones described below (stage 735).

EHFactory caches then loaded script (as an active bag). Email session gets the loaded script and processes it as the follows (stage 740).

The name of the current Active Bag entry is interpreted as IDispatch method invocation. Thus, if script contains "@@Reply" entry, the method "Reply" will be invoked on the EmailSession object (stage 745). This means that EmailSession object has to support IDispatch'able method Reply, otherwise the call will fail . However, if processing the script EmailSession object cannot interpret the text, it invokes Idispatch (stage 750), which provides very efficient, flexible and extendable implementation. The content of the SAFEARRAY, which is an active bag value, is interpreted as the set of either method parameters or text line pairs, where first element is a Property name to set, the second parameter is the property value to set. See script samples below.

If the method fails (e.g. email message sender doesn't match the script criteria), the control is passed to the following selector (stage 755). If the script succeeded, the control is passed to the following selector only if the current selector has "CONTINUE" style. If the current selector has "STOP" style, message processing is concluded. Email manager calls next selector's handler, if any (stage 760). If there is no more selectors to proceed, the Synchronizer is called (stage 765). The peripheral part of email message processing is done.

Peripheral script samples.

A script containing the meta-data entry @@Delete is interpreted as a call to the "Delete" method of the EmailSession object with no parameters.

A script containing the the meta-data entry @@Route Support is interpreted as a call to the "Route" method of the EmailSession object with parameter "Support".

A script containing the meta-data entry @@Reply@Insert "We've just received your message" will be interpreted as call to the "Reply" method of the EmailSession object. The following line starts with @ symbol which means property name to set. The following line contains the property value to set, i.e. text "We've just received your message". In this example, the text will be inserted at the beginning of the replied message's body.

A script containing the meta-data entry @@Reply @Append @ConversationID is interpreted as call to the "Reply" method of the EmailSession object. The following line starts with @ symbol which means property name to set. The following line contains the property value to set, i.e. text @ConversationID. In this case, if property value starts with @ symbol, it means original message's property value. This means that original message ConversationID will be appended to the replied message's body.

The general idea of email peripheral script is that it completely IDispatch-driven. EmailSession does invoke every method from this script on itself, along with the setting/getting the properties, however it does not interprets the script. The methods calling (as well as property access) is done by standard IDispatch mechanism. This will allow the EmailSession object in the future to aggregate any third-party dispatchable object to extend functionality in purely COM manner.

Computer Telephony (CTI)

For detailed information on the Telephony Contact Channel system, please refer to the document TCCS_Architecture.doc and TCCS_EventCases.doc for the architecture and the event use cases.

The primary purpose of TCCS is to implement peripheral level features. It makes no attempt to present the workflow to the caller via the telephony equipment. The peripheral is responsible for delivering calls and providing basic telephony operations to the Agents.

The TCCS is exposed to the core with a startup manager: CtiManager which is a proxy for the FTFTM singleton object

CtiManagerImpl and the IC2Session implementor: CtiSessionManager.

The TCCS is divided into two distinct, yet cooperative sub-systems: the EventRouter and the ServiceProvider.

The EventRouter

The sole responsibility of this system is to gather events from the vendor's middleware, perform first-level filtering and send them to Synchronizer. It maintains a collection of threads for each logged-in agent. Each thread is responsible for delivering asynchronous events into the corresponding agent's workstation. This allows a low-latency at the event collection module and ensures no event is lost while the thread is blocked by Synchronizer.

The Service Provider

This is a collection of one or more service providers that are abstractions to the different CTI peripherals such as the ACD, Voicemail, IVR, etc. Access to these providers is via the CtiSessionManager's IC2Session render().

Factories, Loaders and Meta-data

Data Driven Objects

The entire framework is a data driven system of distributed objects. This means the system itself consists of a small set of core classes, each of which differentiates at runtime into an arbitrarily large number of different objects. For example, the framework consists of a single class, BIO, but at runtime there may be hundreds of different BIOs in the system. Similarly, there is only one of each Step and Operation class, but there will be hundreds or thousands of different Steps and Operations running in a given system. This is what we mean when we say the entire framework is "data driven."

A data driven system needs to have meta-data, and meta-data is stored in a relational database. The meta-data is a critical technological piece of the system, and was designed with no small amount of care and innovation. The meta-data consists of approximately 125 tables normalized to Boyce-Codd 4th form.

Another innovation in the architecture of the framework is that the structure of meta-data is de-coupled from the classes the data drives. In between the two we have Factories and Loaders.

Loaders understand the structure of the meta-data; they generate the SQL needed to read the meta-data, then they denormalize the information into bags - normally, bags of strings.

Factories retrieve these de-normalized bags of strings from the Loaders and they use the information in the bags to build specific types (meta-classes) of objects. Since nearly all framework objects are stateless, most of the factories cache the objects they've created so they can be retrieved immediately in the future. In this case, the caching is particularly efficient, since because the objects are stateless, they don't even need to be copied - all clients merely receive a pointer to the one instance of the object that lives in the cache, and none of the clients knows or cares that other clients may be using the same object at the same time.

Factories, Master Factory and Loaders

When a client object needs a server object, the client object goes to the Factory for the server object and invokes "create". Upon success, the Factory returns to the client object a pointer to the object requested. Otherwise, the Factory returns an error code to the client object.

In most cases, the client's request for an object will be satisfied in the Factory's cache. When this happens, the Factory merely increases the desired object's reference count and immediately returns a pointer to the client, who is expected to release the object when finished with it. This is simple and straightforward; however, things get more interesting when the object requested is not in the cache.

Thread Protection in Factories & Loaders

In this case, the Factory goes to the MasterFactory (a framework-wide singleton) and asks for an object of the specified type. The MasterFactory will go to the appropriate Loader and get a Bag describing the requested object, then return this Bag to the Factory. The Factory will unpack the Bag and use the information therein to build the object requested by the client. If this all succeeds, the Factory gives the client a reference to the object, and it adds the object to the cache.

What makes this interesting is that all this is happening in a multi-threaded environment. While the Factory is busy building object foo, another request for foo might arrive on a different thread. If foo is not already in the cache, the Factory will attempt to build a new foo - so without some kind of cache thread protection, the Factory may not only dedicate multiple thread to constructing multiple copies of the same object, but it will also store these multiple, identical copies in its cache. What is needed is a form of thread protection that allows the Factory to run at optimal efficiency, obviating the need to keep a list of objects that are "under construction" and checking every incoming thread against this list.

The solution to this dilemma is to recognize and take advantage of the fact that the objects being created by the Factory are all data-driven variants of the same COM class object. When the Factory has to create a non-cached object, it can immediately put another COM class into the cache. That way, if another thread comes along asking for this object, that thread can get a reference to the COM class in the cache, instead of having to recreate the object. Of course, since the client got an uninitialized COM class from the cache, if that client tries to use the object it just got from the Factory, it will not work. So we have solved one problem - Factories won't create duplicate objects - but introduced a new problem - objects must be protected so they can't be used by a client until the objects are initialized by the Factory.

To solve this problem, we introduce thread protection in the objects being created by the Factory. Every such object will have an interface that includes an init method. This is used by the Factory to stuff the object with the meta-data that distinguishes it from other objects of the same COM class. Every object will have a special internal semaphore which permits "one writer, multiple reader" access, let us call this the "init lock". In the object's constructor (or ATL's FinalConstruct), it grabs exclusive access to the init lock, which locks its runtime internals from use. In all of its methods, it grabs nonexclusive access (which will block and wait for any pending exclusive access to clear). Thus, any threads that enter the object after it is created, but before it is initialized, will block until the object internally releases its own init lock. Meanwhile, of course, the Factory is busy gathering the data needed to initialize the object. When the factory calls init on the object, the object initializes itself (while these other threads are blocked, waiting), then the object releases the init lock, which opens the floodgates and allows all pending threads to being execution in the object.

In this manner, the Factories are protected from multiple clients asking for the same object at the same time. Also, the users of objects created from the Factory are protected from crashes that would occur if uninitialized objects are used. And all this protection is done without wasting CPU cycles, memory, critical sections or other forced single-threading mechanisms.

Graceful Startup and Shutdown

Startup

All the objects in framework are designed to initialize themselves automatically when they first start up. Because of this, no particular order is needed; the objects simply wake up as needed and start running.

However, MTS will shut down the package hosting the framework objects if that MTS package does not have at least one instance of an object created. has a service that resolves this problem. This service, like other services, starts at system boot. It creates an instance of the framework's key Singleton objects (Translator, Synchronizer, etc.), and holds onto them until the service is stopped. This guarantees there will always be at least one instance of these objects running, which prevents MTS from shutting down the package.

Shutdown

All objects in the system release their run-time resources in their Destructors. However, this is not sufficient to ensure a graceful, orderly shutdown. Some objects keep pointers to other objects, which would require that the some objects be shut down before the other objects (otherwise, the some objects would try to release their pointers to the other objects, but the other objects would already have been destroyed).

Another problem with graceful shutdown is cycles. Often times, objects point to each other in circular reference chains. This is necessary at run time, since a Business Process may have recursive, self-referencing objects - for example, a Step whose Director queues the same Step in a loop, until the Director's Rule determines the loop is done.

The first problem - objects which point to each other - can be resolved by shutting down objects in a certain order. That is, ensuring that the other objects are release before the some objects. The second problem, however, has no simple solution. Something has to break the reference cycle before the system is shut down.

Another factor to consider during shutdown is the fact that MTS constantly monitors the system, shutting down packages that have no objects in use. Here, "shutting down packages" is a euphemism for "killing processes". This presents a difficulty because an MTS package has two kinds of DLLs loaded - those for actual MTS objects, which MTS is managing, and those for other non-MTS related objects, which are being used by the MTS objects. Just because all the MTS objects have been released, does not mean the non-MTS objects are ready to die. But the standard behavior of MTS is to disregard this notion and shut down the package anyway.

So in designing a graceful shutdown, we have three factors to consider:

- When client objects refer to server objects, the clients must be released first. This is not necessarily the order in which the system would unload the corresponding DLLs.
- When objects have reference cycles, the cycle must be explicitly broken before any shutdown can commence.
- When the last MTS object in an MTS package is released, the rest of the objects (all non-MTS objects) must receive a shutdown signal immediately, before MTS kills the package.

<u>Shutdown Implementation</u>

Any coherent shutdown implementation must address the three factors listed above. This section describes how the framework achieves this. To begin, shutdown will occur when the transaction server Service is stopped. Thus, any specific shutdown logic will reside in the service.

Requirement 1: Release Objects in the Proper Order

To address point (1), the transaction server Service will shutdown certain objects in a given order. The order is as follows:

The shutdown logic must first ensure that transaction server is in "idle" state, with all workflows terminated, all contact channel sessions closed, and so on. To this end, transaction server Service may instruct certain critical transaction server components to reject any further client requests. By the time the system reaches idle state, most transaction server objects would have been released. The system then shuts down the remaining objects as described below.

Since MasterFactory caches pointers to Loaders, MasterFactory must shut down before any Loaders shut down.

Since Factories cache pointers to objects, the Factories must shut down before any of the objects in the cache. For example, StepFactory must shut down before any Steps are destroyed.

Since any object may use Synchronizer or Translator, and Synchronizer may use Translator, Translator must be the last object in the system to shut down, and it must be immediately preceded by the shutdown of Synchronizer.

Requirement 2: Break Reference Cycles

To address point (2), reference cycles must be explicitly broken. One way to do this is to require that objects that may have reference cycles to other objects, support a method called deinit(). When this method is invoked, the object in question releases all pointers it has to other objects.

But this in itself will not be sufficient. The caches in which objects are stored must deinitialize their objects, but the caches do not know anything about the objects they are storing - if they did these caches would not be universally reusable. So each cache must be able to determine whether its object support de-initialization, without knowing what kind of objects they are. This is a perfect application for OO interfaces. The framework includes an interface, IShutdown, which contains the deinit() method. When a cache is destroyed, it queries each of its objects to determine whether the object supports the IShutdown method. If it does, the cache invokes deinit() on the object before releasing it. If not, the cache merely releases the object.

Conversely, any object that holds pointers to other objects, and allows reference circles, must implement the IShutdown interface - and in its implementation for the deinit() method, the object must release all of its pointers to other objects.

Requirement 3: Anticipate MTS Shutdown Before it Happens

To address point (3), we must know when the last MTS object has been released, so we can initiate our own shutdown before MTS kills our process. To do this, every object administered to MTS (not just hosted on MTS) will increment and decrement a global reference count. Since they all share the same counter, when it goes to zero all MTS objects have been released and package shutdown is imminent.

Clearly, this cannot happen unless the transaction server Service has been stopped, because it holds references to key MTS objects - their reference counts cannot go to zero until the service releases them. However, the converse is not true - just because the transaction server Service has been stopped and has released its references to the key MTS objects, does not mean those objects will die - indeed, their reference counts may be arbitrarily high, as other objects might (and probably will) be using them.

<u>Implementation Methodologies</u>

<u>Source and UML Directories</u>

<u>UML Model and Code Generation</u>

The code in the framework does not come from Rose 98 code generation. This is because the code generation features of Rose 98 are complex, difficult to use and hard to configure to produce the style of code we desire. It is easier to use the Developer Studio ATL/COM wizard to create a template COM object, then fill in the files.

Notwithstanding this, Rose 98's code generation capabilities are useful for generating code in one's local file system, for checking the effects and implementations of various UML semantics.

Rose 98 provides a system called path maps that are used when generating source code. The path map for the architecture includes a variable that sets the directory into which generated source code is placed. Rose 98 will create subdirectories under this base directory matching the package structure of the model.

One of the problems with the Rose 98 product is how it generates code. When you generate code from a set of classes, the directory structure in which the generated code is placed is ambiguous. If the classes are assigned to one or more components, then the directories will match the layout of the packages in the component view. But the generated code for classes not assigned to any components will be placed into a directory structure matching the package layout of the logical view. Since the component view is a run-time, executable depiction of the system, using its packages to determine the directory structure for generated source code is an inscrutable design oversight. And while building IDL and .H files from the logical view is pretty straightforward (though [in] and [out] parameters for the IDL cannot be specified by the user), generating CPP files from the logical view is a burdensome task with often unpredictable results.

The ideal tool would always use the logical view for generating IDL, .H and .CPP files, using a directory structure that matches the package structure of the logical view. Whether any classes are assigned to components is irrelevant for this task. The component view should be used for generating project files and makefiles. Though none of the tools has reached this level of simplicity, Rose 98 appears closest to this goal.

Directory Layout of the Framework Source Code

The directory structure of the source code is modeled after the package structure of the framework UML. OOA/D paradigms such as dependency inversion (package A owns the interface that package B implements), package dependency mapping and strict interface/implementation separation are designed into the model and reflected directly into the structure of the source code and build procedure.

Many packages own interfaces, but don't own the implementation - such as package BusinessData, which owns interface Iproperty. These packages live physically as directories in the source code filesystem. The interface lives physically as an IDL file. This IDL file contains only an interface description. It does not contain any class or type library information.

The directory will also contain a project or makefile that will compile all the interfaces it owns. Other packages that define implementations for these interfaces, depend on this package and import, include or reference the interface files.

Since the directory structure of the source code matches the package structure of the UML, and the dependencies of the source code match the class dependencies of the UML, the order in which objects are built is straightforward and well defined. This order is reflected in the build procedure, the release cycle, and the very organization of the engineering department.

UML Model, Rose 98 Subunits and Logical Packages

The Rose framework UML model is broken into subunits, one per package. Every package in the logical view comes from a separate subunit. Each subunit is stored as a separate Rose .cat file on the disk. The master Rose .mdl file maintains references to each of the subunit files so that the entire model can be loaded and edited seamlessly as if it came from a single monolithic file.

Since the packages represent relatively autonomous units whose dependencies are already tracked and analyzed, they embody a coherent and logical decomposition for the responsibilities of the engineers in the development organization.

Counter-intuitively, packages are not broken down into functional areas. Rather, their partitioning is based on dependencies. Remember that if class A uses class B, then A depends on B. Thus the package that contains A depends on the package that contains B. Since high level code normally invokes lower level code, this inverts the desired dependencies because we want the low level code to depend on the higher level code.

To get around this, we can use OO languages to reify the interface of the lower level code and move it to the higher level package. Thus the lower level package is forced to implement an interface that the higher level package owns. This reinverts the dependencies to what we wanted - lower level code depending on higher level code - while keeping the desired dependency structure of the packages.

Logical UML Interfaces to Physical COM Interfaces

COM interfaces are completely static, thus do not have the flexibility of C++ interfaces. This leads to a slough of potential implementation problems, each of which is addressed below.

Aggregation vs. Inheritance

Though objects and interfaces are both designed as classes, in the Architecture they have been separated as clearly as possible. Interfaces are modeled as abstract classes containing no properties and consisting entirely of pure virtual functions. Objects are modeled as concrete classes that implement one or more interfaces. Most of these interfaces also have a related object that provides a simple, yet complete implementation of the all the functions in the interface. This setup has two key benefits.

First, it creates a clean programming architecture where interfaces are inherited, yet behavior is aggregated. This resolves the ambiguity behind the overloaded use of inheritance in statically typed languages (like C++). It also ensures a smooth and intuitive deployment in DCOM. The interfaces can be described with IDL and the objects can be deployed as DCOM objects with the simple addition of an Iunknown implementation.

Second, separating objects and interfaces dramatically cleans up the UML. Any class relationship except inheritance requires properties in at least one of the related classes. Thus the only relationship in which interfaces can take part is that of inheritance. This means it is both intuitive and precise to allow other relationships, such as usage and aggregation, only across concrete objects.

Interestingly, a concrete class may send messages to an interface - though the reverse is not true. This is obvious when one considers that an interface is a set of functions and sending a message is really just calling a function. This is asymmetric because sending a message is behavior, so no interface can do it.

Here are the resulting rules:

Inheritance is the means for interface reuse.

Aggregation is the means for behavior reuse.

All interfaces must consist entirely of pure virtual functions and have no properties.

All objects must implement at least one (and possibly many) interface.

Here are some interesting corollaries to the above rules:

An interface may not be derived from a concrete class, nor may it send messages, nor may it aggregate any other class.

An object may not be derived from other objects. Where an object extends or specializes the behavior of another object, it must aggregate that object and inherit its interface. Of course, it may inherit the interface publicly or privately.

Aggregation by reference must be aggregation of interface.

Aggregation by value must be aggregation of an object.

Multiple Interfaces, One Object

One of COM's fundamental uses is separation of interface and implementation. Of course, this leads immediately to a many-to-many relationship between interfaces and objects - one interface implemented by multiple different objects, and a single object that implements multiple different interfaces. Because ATL was designed mostly to simplify COM programming, it seems to be geared for a single-interface world where every object has exactly one unique interface. Thus, implementing true interface/object separation in COM using ATL becomes a deceptively subtle task.

One Interface, Multiple Objects

The first (and simplest) case is when a single interface is to be implemented by many different objects. A good, clean example of this in the framework is properties.

There is one property interface IProperty. There are many property objects IntProperty, StringProperty, etc. No property object has an interface of its own. Each property object implements IProperty as its sole public interface. This way, a client can talk to a bunch of different properties without concern for their individual types (if the client wants to know the type of a property, it can use the getType method in IProperty).

There is a single IDL file, IProperty.idl, which lives in the BusinessData directory (because the BusinessData package owns the IProperty interface class). This IDL file defines only the interface. It does not define any classes or type libraries. Thus, it produces only an IID when it is compiled - no CLSID or type library.

To make all this happen, the BusinessData directory has a project file called BuildIDL.dsp, which runs a makefile called BuildIDL.mk, which compiles the IDL files. This is true in general of all packages that own interfaces.

Each property object lives in a subdirectory of BusinessData/BDProperty and has its own IDL file. This IDL file merely imports IProperty.idl for the interface definition, and adds its own class and type library. Thus, when the IDL for each property object is compiled, it uses the same IID for IProperty, but builds a new type library with its own CLSID.

Internally, the class that implements the property object is derived from the IProperty interface, and its COM interface table includes IProperty and IDispatch.

Each property object has its own project file, and all this happens automatically when the property is built.

<u>One Object, Multiple Interfaces</u>

A single object can implement multiple interfaces in two fundamentally different ways. First, it can implement multiple independent, unrelated interfaces in parallel. Second, it can implement a single interface that itself is derived from another (or more) interface. These two cases are described separately below.

<u>Multiple Parallel Interfaces</u>

This means a client can QI for any of these interfaces, and though they all go to the same object, the interfaces themselves aren't related. A good example of this is the CoreObject object. It has two interfaces, ICoreObject and ICoreObjectPrivate. These interfaces are separate and unrelated.

Both of these interfaces are owned by the BusinessServer package, so the IDL files for each (ICoreObject.idl and ICoreObjectPrivate.idl) live in this directory and are compiled by the BuildIDL.dsp project. Each is a dual interface derived from IDispatch.

The CoreObject object does not have its own interface. It implements these two interfaces owned by the BusinessServer package. As described in the previous section, it has its own IDL file CoCoreObject.idl, but this IDL file does not define an interface. Rather, it merely imports ICoreObject.idl and ICoreObjectPrivate.idl, and defines its own class and type library. In its IDL, it declares support for both interfaces, but picks one as the default (ICoreObject).

The class that implements the CoreObject is CCoreObject. It is derived from both ICoreObject and ICoreObjectPrivate (using the IDispatchImpl ATL macro for both) and has both interfaces in its ATL/COM interface map.

<u>Single Derived Interface</u>

In many cases a design calls for interface inheritance. Here, a single object must implement an interface Ic that is derived from a parent interface Ip. Here the following rules apply:

Interface Ic must include all the methods in Ip, as if they belonged to Ic. Thus the client can use the entire set of methods encompassed by Ic and Ip together, in the single interface Ic.

The object must also implement Ip directly, so a client can treat it as if it were an object of type Ip. This is the type checking side of inheritance - an object that implements Ic must be able to masquerade as an Ip.

An example of this in the framework is the ISingleBag interface, which is derived from IBag. First, the ISingleBag interface inherits all the methods in the IBag interface, so the client can use it without needing an IBag. But also, any object that implements ISingleBag must also implement IBag, too. This way an object that supports ISingleBag can be passed around and used as if it were an IBag.

To implement this, the top level interface in the inheritance chain (here, IBag) is derived from IDispatch. All its derived children are derived from it. For example, ISingleBag is derived from IBag. Because COM does not support polymorphism, this means ISingleBag cannot also be derived from IDispatch.

The object that implements ISingleBag must also implement IBag. The fact that the ISingleBag already includes an IBag is not sufficient, because in order for the object to be fully polymorphic to an IBag it must present this interface alone to clients that don't know about ISingleBag interfaces. Of course, whether the client gets to the object through IBag or through ISingleBag is irrelevant because both interfaces hook up to the same internal functions.

The IDL for the SingleBag object imports ISingleBag.idl for its interface definition. It declares a coclass that supports both ISingleBag and IBag with the former as its default interface. When the IDL is compiled a type library is generated.

The class that implements the SingleBag COM object (CSingleBag) implements three interfaces: ISingleBag (its default), IBag and IDispatch. Class CSingleBag is derived from ISingleBag alone, but has all three interfaces in its ATL/COM interface map.

<u>Parameterized Classes (Templates)</u>

The difference is realized mainly in C++ templates. One can in C++ define a single template interface class such as IBag<class T>, which can a bag of anything. But in COM, you'd have to have a whole slough of separate, yet similar interfaces such as IBag<integer>, IBag<string>, etc. The question is how to manage this complexity.

<u>Separate Interfaces, Separate Objects</u>

From a design perspective, the most obvious approach is to have a separate COM object for each edition (and interface) of the bag. For example, there'd be an interface IBag_integer with its own IID, implemented by a class CBag_integer with its own CLSID. The same would happen for strings, floats and all other data types. Each would have its own independent CLSID and IID. The client would have to pick the type of bag it wanted by selecting the appropriate CLSID in CoCreateInstance(), then pick the correct interface in QueryInterface().

This makes the implementation far more complicated than the model, which has a single interface and a single object. And since the bags treat their elements as opaque objects, merely pointing to them without ever looking inside, the complexity of this implementation - while necessary to satisfy the physical limitations of COM - is logically unnecessary.

Separate Interfaces, One Object

We can reduce some of the complexity seen by the client by building a single class that implements all the (slightly) different IBag interfaces. In this scenario, the client simply creates a CBag object, then uses QueryInterface() to get the interface it wants.

The CBag actually only contains a void pointer, so it is very lightweight and created instantly. When it receives its first QI request (probably though not necessarily from CoCreateInstance), it creates the instantiated class for the template that is appropriate to the requested interface, and keeps a reference to it with its void pointer. From that point on, it satisfies requests to the interface with the class.

For example, suppose the client first QIs for an IBag_integer interface. The CBag object does something like this:

```
p_myBag = new(IBag<int>);
e_myType = integer;
```

From that point on, it delegates requests on the IBag_integer interface to p_myBag. And it refuses further requests to QI, unless they are for the IBag_integer interface.

The object has morphed itself to an integer bag. If the client wants a bag of strings, it may create a new instance of CBag and QI this new bag for the IBag_string interface.

One Interface, One Object (using VARIANTs)

A third solution to the problem of COM interface inflexibility is to use VARIANTs. Here we have a single IBag interface and a single CBag implementation. Every bag is a bag of VARIANTs.

The key problems with this scenario are the following:

- How to stuff pointers to the VARIANTs
- How to determine what kind of objects are being stuffed into the bag (so the Bag can cast them back to the appropriate types before returning to the client)
- How (or whether it is necessary) to ensure that the client is passing objects of the same type into the bag

If we make the client responsible for casting its objects to and from the VARIANTs, our problems are solved. And since it is quite straightforward to do this, this would not be any great obligation on the client.

One Interface, One Object (using Iunknown)

Having recognized that this problem pertains mainly to bags and iterators, and that bags and iterators don't know anything about the objects they use, we can further optimize the implementation.

When a client wants a bag of objects, it doesn't really want a bag of objects. Rather, the client uses only particular interfaces of the objects in the bag. And the client is unconcerned with whether the object sitting behind the interface supports other interfaces. Thus we can implement a bag as a bag of interfaces.

Recognizing that every COM interface is derived from IUnknown, we can declare that every bag is a bag of IUnknown interfaces. This leaves the client free to cast or QI the

IUnknowns it gets from the bag. It also enables the client to store bags of heterogeneous objects.

Thus, bags and iterators - and parameterized classes in general - are implemented in COM and in the physical deployment of the Architecture as classes that operate on references to IUnknown interfaces.

This leaves us with a client implementation that looks something like this:

```
interface foo;
p_myBag = new(IBag);
p_myBag->add((IUnknown *)foo);
```

<u>Overloaded Functions</u>

Standard C++ enables one to provide many different versions of a function, all with the same name. As long as the functions' formal parameters are different there is no name collision. In other words, the signature of the function is its name combined with its formal parameter list.

In IDL, this is not the case. A single IDL file defining a single interface cannot contain two methods of the same name, even if their formal parameter lists are different. This is an annoying headache for developers who enjoy the liberty of overloading function names to provide simpler, more intuitive interfaces.

Certainly, one could put each overload of the method into a different interface and force the client to get to each by invoking QueryInterface() on the server object to get the interface that contained the version of the method the client needed. This is a bad choice for many obvious reasons, not the least of which are the fact that the client would have to "know" which interface contained the version of the method it needed, and that it adds useless and unneeded complexity to the system. So how does one get a single interface on a server COM object, and see within that interface overloaded versions of a function? We have discovered two methods. While one is an evil hack, the other may be useful.

Overloading by Interface Inheritance

Suppose we want the client to see an interface called foo which contained three different overloaded versions of method bar(). One useful way to do this is to build a total of four interfaces. First, we have interfaces foo1, foo2 and foo3, each of which defines a different version of method bar(). Finally, we have an interface foo which is derived from the first three interfaces. This final foo interface would contain all three overloaded versions of bar().

Actually, there is one more piece of complexity here. IDL is not able to express polymorphism, so we would have to build a chain of inheritance where foo is derived from foo3, which is derived from foo2, which is derived from foo1.

At the end of the day, a client object uses the single interface foo which contains all three overloaded versions of bar(). The client is blissfully unaware that these versions were inherited from other interfaces.

Aggregation and Containment

Structurally, objects collaborate using aggregation and containment. These are similar, yet different ways in which objects are grouped together or point to each other.

Containment describes the general case of component A acting as the client to another component B. Whenever A contains B, A either contains a B (by value) or a pointer to a B (by reference). The client of object A does not normally know or care whether A contains B by value or by reference. What is more important is whether A exposes to its client the interfaces that B supports, and if so how it does this.

There are a few different ways to do this.

Single Interface (containment)

The simplest case from the client's perspective, A can provide a single interface to its client, and privately use object B to implement parts of that interface. In this case the client does not even know that B exists. This is a case of containment.

Multiple Interfaces (containment and aggregation)

In a case more complex from the client's perspective, A can provide multiple interfaces to its client. The client must query A for the different interfaces it wants to use. It appears to the client that A supports all these interfaces, even though A is really deferring the implementation for some of them to B.

If A does this by directly passing B's interfaces to the client without intercepting or modifying them, this is called simple aggregation.

If A does this by intercepting calls to B's interfaces and wrapping extra functionality around them, this is called containment. The difference between this and the first case of single interface containment is whether the client is aware of the different interfaces. Here the client must query A to get a new interface, where in the first case the client invokes all functionality from a single interface on A.

In UML, we tend to use the term aggregation loosely, to identify all forms of containment.

"Public aggregation" may refer to the use of simple aggregation or the use of single interface containment. In general it means the client can access the functionality provided by the aggregated object.

"Private aggregation" refers to the use of containment. Private aggregation is often used as part of a Strategy design pattern.

UUIDs: Interfaces, Type Libraries and Objects

The COM standard identifies objects with UUIDs: Universally Unique Identifiers. Every COM object has several UUIDs related to it:

One class UUID: Known as a CLSID, this is a UUID that identifies the COM object itself.

One or more Interface UUIDs: Known as an IID, this is a UUID for the interface. Since every object can implement many interfaces, there may be many IIDs associated with the object.

One type library UUID: Every object has a single type library and the type library has a UUID.

Every COM object implemented as an in-proc DLL has a single file named dlldatax.h, which defines the entry points standard COM requires in the DLL that hosts the COM object. Like the rest of the files in the source code, this file has #DEFINE protection to prevent redefinition of the symbols in the file when it is #INCLUDEed multiple times. For precision and redundancy avoidance, the #DEFINE protection of each dlldatax.h file has the CLSID of the COM pertinent object.

Threads, State and Object Recycling

The framework is intended to scale at a level supporting hundreds of clients on a single transaction server. Toward this end, certain key objects are designed for instance reuse. For example, multiple different clients running the same workflow are all represented as different threads running inside the same actual instance of a single workflow object in the transaction server.

Objects with reusable instances must be carefully designed. For instance, the above example implies that the workflow cannot maintain a pointer to its context. This would tie an instance of the workflow to a particular context. Instead, the context must be passed to the workflow object. Thus, multiple different clients each running the same workflow can pass a different context. That way, each thread running this instance of the workflow object has its own context.

This also requires objects to be thread safe. At any given moment, there will probably be multiple threads walking through different areas of the object. Judicious use of critical sections is required for objects, such as bags, where what one thread does must be seen by all the others. We must avoid thread lockouts, destruction of the object's state by two competing threads, and other thread synchronization problems.

MTS, Factories and Object Reuse

Objects in the framework are meta-data defined. At run time, COM objects are instantiated, then initialized to become appropriate metatypes. For example, a "BIO" might be instantiated, then init() used to make the BIO a customer. Thus the "type" of an object is defined on two levels:

First is the object's COM type - its CLSID. This object is a merely a shell or husk that can at run time become any particular metatype.

Next is the object's type (or metatype) - its ClassName. This defines the metatype of the object.

When MTS reuses an object, it throws away the object's "state" and reuses the empty husk (or COM type). If two MTS objects have the same CLSID, then MTS may "hot swap" them - that is, use them interchangeably to resolve proxy objects of different clients.

Hot swapping the husk of an object across multiple clients does not always conserve resources. It requires the object's current state to be thrown away or at least detached from the husk of the object, and it requires the new state to be glued to the husk. Hot swapping saves memory by spending CPU cycles. Whether this improves scalability depends on the nature of the object. The example below depicts two extreme cases that together emphasize the following rule:

MTS provides immense benefits for big objects with small state. But small objects with big state are an MTS nightmare.

Hot Swapping/Scalability Example

Case 1: Big Husk, Small State

Consider an object whose husk is gigantic compared to its state. An example would be an object with a lot of complex methods (lots of code) that supported many interfaces (many vtables), but with only a single integer data member.

When MTS hot swaps this object between two clients, it obviates the need to duplicate the object's methods and interfaces. The only price is that a single integer must be saved somewhere and restored later. Scalability is greatly improved because we save a vast amount of resources at the cost of a small number of extra CPU cycles. This is the ideal case for MTS.

Case 2: Small Husk, Big State

Consider an object whose husk is tiny compared to its state. An example would be an object with only two methods (say, get and set), but with huge amounts of complex data structures.

When MTS hot swaps this object between two clients, it saves only the husk of the object, which in this case is an insignificant savings. And it requires detaching and saving a huge, complex data structure. We have crippled our system by wasting thousands of CPU cycles merely to save insignificant amounts of memory.

Object Initialization Stages

Take a snapshot of all the (thousands of) objects instantiated in a running transaction server and each will be in one of three stages:

- Stage 1: Raw, Stateless

  Here, the object is instantiated, but not initialized. It has no metatype and is unusable. ClassName is nondescriptive ("BIO," "Step," etc.) and InstanceName is NULL. The object has no metatype.

- Stage 2: Empty, Static State

  Here, the object is instantiated and initialized. ClassName is specific ("BIO.customer," "Step.GetCustName," etc.) but InstanceName is still NULL. The object has a metatype and is usable.

- Stage 3: Full, Dynamic State

Here, the object is instantiated, initialized and has been used. Its ClassName from stage 2 remains, and it may have an InstanceName. The client has used the object in a way that gave it dynamic state - e.g. set(), etc.

In stage 1, most objects are fairly small. The exceptions would be objects that have a large number of complex method functions. Certainly, a given object is smaller in stage 1 than it is in any other stage. Also, in stage 1, the objects are not very differentiated. In fact, any two objects of the same CLSID could be hot-swapped freely with no negative effects. Unfortunately, they are so small that the benefits of doing this would not be great.

Entering stage 2, the object grows. Normally, it grows immensely, gaining something like a large multiple or an order of magnitude in size. The objects are also more differentiated, too. But there still is great opportunity for hot swapping. Two BIOs may be different (say, "customer" and "account"), but all "customer" BIOs are still the same because they have no dynamic state - retrieve() and set() have not been used. We would gain great advantages from hot swapping objects in stage 2, but MTS won't do it for us. This is because MTS will hot swap any two objects with the same CLSID, and though "customer" and "account" have the same CLSID, they cannot be hot swapped because their metatypes are different.

Entering stage 3, the object either does not grow at all, or it grows insignificantly. But it breaks free from all other objects, as differentiation is complete and every single object is different. Hot swapping can be used only if the object's dynamic state can be detached and saved, then restored later when it's hot swapped back in.

Reuse at Stage 1, Stage 2 and Stage 3

Here we describe how to achieve maximum scalability through object reuse at each stage of an object's lifetime.

Stage 1 Reuse

In stage 1, all objects of the same CLSID are reusable (hot swappable). Since this is the default behavior of MTS, administering the objects to MTS does this transparently.

Stage 2 Reuse

In stage 2, all objects of the same metaclass are hot swappable. Objects with the same CLSID cannot necessarily be hot swapped because their metaclasses may be different. Since MTS views the metaclass information as if it were dynamic state, MTS cannot be permitted to hot swap objects in stage 2.

To get around this, we need to detach (and save) the current metaclass information just before MTS hot swaps the object, then swap in the (saved) metaclass information for the client whose context the object is getting swapped into.

To get MTS to hot swap objects in stage 2, we need to do two things:

Find a way to remember which metaclass information to use when the object is hot swapped into a new context.

Find a way to recreate the appropriate metaclass information for the object.

The second item is easy. The framework has already named the object's metaclass information, and we already have a factory that can provide it instantly whenever we ask for it by name. In fact, since the metaclass information is now shared across all objects of a given metaclass, the factories get much simpler and the metaclass information becomes a Singleton.

So now all we have to do is store the name of our metaclass somewhere, just before MTS hot swaps us out of existence.

The natural place to do this is in the MTS context. Whenever we get hot swapped back in, we are guaranteed to have the same context we had just before we were hot swapped out. If we could store our metaclass name in the context, the problem is solved. But the MTS context is not easily extensible.

Instead, what we can do is use MTS shared property groups. This is a global storage area provided by MTS, where we can store any data (our metaclass name) and access it by name. Now we need to have a name we can remember after we've been hot swapped in and out. Anything in our context that is unique will work. But remember that while all the objects owned by a particular client have different contexts, they all share the same session ID. So the context's session ID won't do the job.

What we can do instead is to use the address of our context as the name of the location in the shared property group where we store our metaclass name.

Stage 3 Reuse

Objects in Stage 3 are inherently not reusable, since their dynamic state is not sharable to any other object. Instead, what we do is save and restore their stage 3 dynamic state as needed. Since this state is much smaller than the total size of the stage 2 object, we have succeeded in taking advantage of MTS to share what can be shared - that is, to maximize scalability.

Implications of Staged Object Reuse

Now that we have a plan for staged object reuse under MTS, let's look at how to accomplish it. The framework's behavioral objects are designed to have no stage 3 (dynamic) state information, so the plan is quite straightforward But objects that are used to model state - such as BIOs, are more complicated to manage.

Behavioral Object Reuse

Use the method described above: MTS manages the stage 1 object, and factory objects coordinate with MTS to provide stage 2 reuse. There is no stage 3 state, so we're done.

Data Object Reuse

This is the same as the case for behavioral objects for stages 1 and 2. But data objects have a stage 3 dynamic state which must be assumed to be globally unique. To reuse these object successfully we must do two things:

- De-couple the stage 3 (dynamic) state of the object from its stage 2 (metaclass) state.
- Find a way to save the decoupled stage 3 (dynamic) state just before MTS hot swaps us, and to restore when we are swapped back in.

Container Objects

The framework makes extensive use of container objects. To satisfy constraints including but not limited to performance, scalability and thread safety we have designed our own container objects. These objects have interfaces owned by the DataTypes package in the BusinessServer package.

Operations on a container may change the state of the container - e.g. addItem, dropItem, etc. - or may not - e.g. getNext, getData, etc. A single container may have many different clients, some of which are adding and dropping items, others only iterating over the container. And each of the latter clients may want to iterate in different ways - some forward, some backward, etc.

The first step to satisfy efficiently these requirements is to implement each container as a Bag that may have many access gateways.

Passive Bags vs. Active Bags.

A passive bag contains items, but it does not own the items it contains; thus, it does not duplicate or destroy the items. A passive bag is merely a collection of pointers to items. The bag increases by one the reference count of each item it contains, and decreases the item's reference count when the item is removed from the bag. Ultimately, it is up to the item to decide when to destroy itself based on the reference counts maintained by its clients. The passive bag is only one of many possible such clients. To access data in a passive Bag one needs an Iterator. Several Iterators can be created for multiple clients to access the Bag's contents independently and simultaneously.

An active Bag allows accessing items directly, through the Bag's interface. The capability is orthogonal to the Bag's internal implementation. The bag may contain its items directly, or it may contain just pointers to the items. It is up to developer which kind of active Bag to use, depending on performance, memory and other considerations. A bag consisting of pointers implies an extra level of COM indirection, while a bag consisting of data implies possible duplication of data. To access data in an Active Bag one creates an Access Gateway; several gateways can exist simultaneously inside a bag, providing independent and simultaneous access to the bag's items.

<u>Active Bags and their access gateways</u>

The active bag groups its items and provides a means to add and drop items from the group. Internally, the active bag uses SAFEARRAY to maintain the collection of items, but the client of a bag remains blissfully unaware of the implementation. The client passes an item to the bag with add(), and specifies an item with drop(). The client can specifically addToEnd() or addToStart(); the bag adds to end by default.

An access gateway provides a means to visit the items in an active bag. The bag creates a gateway for a client, by the bag's CreateAccess() method, and drops it by DropAccess(). Every gateway has the HANDLE, assigned at Create time, which uniquely identifies the gateway among others in the bag. The client then calls Bag's methods, referring to the gateway handle: first(), next(), last(), prev(), isDone(), getFirst(), getNext(), getLast(), getPrev(). While the gateway points to an existing item, the methods return transaction server_S_OK;

when current gateway is driven outside the Bag's SAFEARRAY, the methods return transaction server_S_FAIL, so that visiting a Bag always SUCCEEDs. Bag provides additional functionality, such as get(), getAtIndex().

Although an active bag can change its contents, while some gateways are accessing the Bag, there is no arbitrary limit to the number and type of gateways that can simultaneously visit a single bag. Many different clients can all independently and simultaneously iterate over a bag in different ways without stepping on each other's toes - as long as the Bag doesn't change its contents.

Passive Bags and their Iterators

The passive bag groups its items and provides a means to add and drop items from the group. Internally, the passive bag uses nodes to encapsulate and join the items it contains, but the client of a bag remains blissfully unaware of nodes. The client passes an item to the bag with add(), and specifies an item with drop().

An iterator provides a means to visit the items in a passive bag. The basic iterator interface (Iiterator) is comprised essentially of first(), next(), isDone() and get(). Of course, what next actually means is up to the iterator in question, but it guarantees to visit every item in the bag before isDone() returns TRUE. Different types of iterators may provide additional functionality, such as prev(), last(), or perhaps getAtIndex().

Because nothing an iterator does can ever change the state of its bag, there is no arbitrary limit to the number and type of iterators that can simultaneously visit a single bag. Many different clients can all independently and simultaneously iterate over a bag in different ways without stepping on each other's toes - and if the Bag changes its contents, it immediately updates (and resets) all outstanding iterators.

Though iterators visit all the items in a bag, they know nothing of bags; they don't even know that bags exist. This is critical because bags already know about iterators, so if the converse is true we have a package dependency cycle. So how do iterators visit the items in a bag without knowing about bags? Both iterators and bags know about nodes. The rest is carefully designed interaction.

Passive Bag/Iterator Interaction

Where do iterators come from? They can't just be instantiated out of the blue, because then you'd have to tell them what bag over which to iterate, and they don't know what bags are. The solution is to have the bag create its iterators. When bag client wants an iterator over the bag, he tells the bag to CreateIterator(). The bag creates an iterator, glues it to the bag's first node (item), and passes a reference to this new iterator back to the client. When the client is done with the iterator, the client uses the DropIterator() method on the bag to destroy it.

This scenario is useful for several reasons:

The iterators don't need to know about bags. The iterator begins life glued to a node, it doesn't know the node came from a bag. The bag can glue it to a node because the bag knows about nodes. The bag's client cannot do this because the bag's client doesn't know about nodes.

Only the bag knows what kind of iterators are pertinent. For example, a SingleBag (singly linked list) cannot produce an ArrayIterator (random access, indexed iterator).

When a bag changes, all its iterators must be reset. Since only the bag knows when its state changes, only the bag is qualified to send messages to the iterators. And since the iterators are created from the bag, the bag can keep track of them.

Implementation Subtleties

Earlier we mentioned that every bag maintains keeps track of all the iterators it has created for its clients. It keeps track of these iterators in another bag. Thus, every bag contains another bag. This creates obvious problems when we want to create a bag. For example, suppose I create a bag of rocks. The bag of rocks creates its bag of iterators. The bag of iterators, itself being a bag, creates a bag of its iterators. This continues ad infinitum, our original request for a bag of rocks cascading into an infinite chain of bags that must be created.

Since no bag ever needs a child bag of iterators until one of the main bag's clients asks for an iterator, creation of the child bag of iterators is deferred until the point when a client asks the bag for an iterator. As a result, the infinite cycle is broken by avoiding the need for the constructor of the bag to create a child bag.

Now, when a client asks for a bag of rocks, it is created and returns. No other bags are created. When the client asks for an iterator on the bag of rocks, it creates a child bag to store the iterator, and returns to the client a reference to this iterator. The client remain blissfully unaware of the child bag the bag of rocks has created to keep track of the iterator it created for me.

Now suppose the client asks the bag for another iterator and the client then gives ithe iterator to his neighbor. Internally, the bag creates a new iterator and adds it to its child bag, which now contains two different iterators.

Finally, suppose the client takes a rock out of the bag. The bag will send all its outstanding iterators a message to reset themselves. To do this, the bag of rocks must iterate over its child bag of iterators. So the bag of rocks asks its child bag for an iterator. The child bag creates a grandchild bag to keep track of this iterator, and returns a reference to the new iterator to its client, the bag of rocks.

As shown in Fig. 8, the chain of bags is three levels deep: the main bag 810 (of rocks), which has a child bag 820 (of rock iterators), which has its own child bag 830(of iterators over rock iterators).

When the bag of rocks 810 is done iterating over its iterators, telling each to reset itself, it may either keep or discard this meta-iterator. But the child bag 820 (of rock iterators) will never have any need to iterate over its own child bag. Thus, the hierarchy will never reach level 4.

Even if it did, this would not cause any problems. Deferring the creation of the child iterator bag to the time the first iterator is created has solved the problem.

Free Threaded Array Bags

Yet another type of bag is the Free Threaded Array Bag. It is a special kind of passive bag, designed and optimized for raw performance under the following special circumstances:

The FTArrayBag is FT/FTM, unlike standard passive bags which are apartment threaded.

Every item in the bag must be FT/FTM (free threaded with a free threaded marshaller), because it does not intern (cookieize) the pointers to its items.

The FTArrayBag's contents should be static - set up once, and never changed. That is, once the bag is set up, items should not be added or removed.

The FTArrayBag's iterators are apartment threaded. This ensures that they can be instantiated extremely quickly.

These requirements are precisely those of the behavioral model, particularly with Sequences and Directors.

Since the entire behavioral model is FT/FTM, it is critical to have FT/FTM bags to use as building blocks for composite objects. Otherwise, the objects have to intern pointers to apartment threaded bags, which introduces a single-threading bottleneck as multiple clients all have to walk single file through the apartment in which the FT/FTM composite object was first initialized.

Measured in real-world terms using load tester, introducing the FTArrayBag improved performance and scalability by a factor of 2:1.

LiteBags

A LiteBag is the lightest of the container structures, based directly on the lightest COM-compatible aggregate: a Safearray. A linear Safearray of Variants is used to store a set of string name — Variant value pairs, even elements holding item names and odd elements holding item values.

A class, called CLiteBag, manages the Safearray and exposes it either as an associative array (inserting and retrieving items by name), a regular array direct-accessed by index, or a combination of both. In the case of regular array, multiple items with the same name are permitted. Of a set of items with the same name, associative access reaches only the first one. Note that LiteBag is not intended as a recursive container. (Although a Safearray of VT_VARIANT can in principle contain another Safearray as the value of its element, managing such an arrangement through LiteBags is quite cumbersome.) LiteBag has neither Access Gateways nor Iterators: concurrency control is entirely up to the client. The Bag grows to accommodate new data, but does not automatically shrink. Consequently, all numerical indices to live items remain valid until Shrink() is called.

Much like CComBSTR and CComPtr, LiteBag is not a COM object but a class; to transmit its data across COM boundaries, the sender should detach the underlying Safearray, and the recipient should attach the array to (its own local instance of) LiteBag. A companion class, called CLiteBagView, exposes the underlying Safearray in the same way as CLiteBag but does not control its lifetime. This is useful for input parameters that you do not want destroyed at the end of the function call.

Not being a COM object, LiteBag is equally suitable for all threading models. The implementation is made threadsafe (even though Safearray intrinsically isn't) by critical section calls around every method. Capturing an available critical section incurs only a small overhead (2us on average).

Security

The goal of security in the transaction server is to meet the needs of the applications by using as much of the built-in capabilities provided by the Microsoft server platforms. We do not want to implement our own security system.

Transaction server Security Requirements

Capabilities Of The Platform

NT Security

Windows NT 4.0 provides a security model that is based on security domains. Users, Groups and other resource accounts) can be added to domains. Domains can trust other domains (i.e. they trust the users logged onto those domains).

The model used is a challenge-response model where passwords are never sent across the network but instead keys are generated based on the password supplied and a key obtained from the domain controller. This ensures a secure way to authenticate.

Server processes can impersonate their clients' security in order to access resources on behalf of the client. This provides a powerful way for application servers to integrate NT security into their own transactions without doing too much work.

A limitation of the current NT security, due to the fact that passwords are never sent across the wire, is that when a server impersonates a client, it cannot call another server on a different physical machine and have that server impersonate the same client. This means that client impersonations cannot hop machines. (This limitation will go away with Windows 2000 where they will have the Kerberos security implemented).

Another limitation of NT security is that an NT domain can have at most 64,000 accounts in it. So, if a customer has more than approx. 60,000 users, they will need more than one NT domain to hold all their user accounts. This can get complicated in terms of administration as trust relationships need to be made between domains. This limitation is not too serious for now because Windows 2000 promises to solve this problem - worst case, a customer will have to administer multiple NT domains.

IIS Security

If standard NT authentication is used from a browser, the IIS logs in on behalf of its client to the NT system and then after that all transactions occur in the security context of its client. Unfortunately the security context does not appear to survive a process hop. Most of transaction server therefore does NOT actually run under the browser's NT account, but under its own account that is the same for all users. Username of the browser account is stored in the Context and may be used by transaction server to control access to business objects.

An application implements authentication as follows:

1. Workflow fires ActAuthenticate and suspends, in much the same way it would suspends for a Render operation. ActAuthenticate calls IC2Session::Authenticate() and then suspends.

2. HttpSession returns a special code to OtsGateway.asp.

3. OtsGateway.asp returns '404 Unauthorized' to the browser.

4. Browser retries the hit with new credentials. This could be the 'implicit' credentials derived from the client's NT login, or something the user had typed into a dialog box.

5. HttpSession retrieves the new username and passes it to ISynchronizer::setUserClue(). Username (e.g. 'bslutsky') is

the only identification token used by. Domain name is not retrieved, and neither is the password.

6. ActAuthenticate resumes and populates CharacterUser BIO based on username supplied. If Populate returns more than one row, Authenticate picks the first row. If Populate returns no rows, ActAuthenticate fails.

7. The workflow checks ActAuthenticate return code and if ActAuthenticate failed repeats from step 1.

8. Windows NT appears to limit the number of login dialogs it is willing to pop up. After 3 failed attempts (counting both the attempts rejected by ActAuthenticate and invalid NT accounts that never reached ActAuthenticate), the browser does not pop any more dialogs. This leaves the workflow suspended and (the way the app is currently set up) leaves the user staring at a blank screen. Not elegant, but secure. This is the best we can do since NT authentication gives us no way to determine whether the user has run out of attempts, or still staring at a dialog box, or navigated out of a long time ago.

<u>Secure delivery of Content through IIS</u>

Content items, as any resource, can be secured in one of two ways:

A. Security characteristics are attached to each Content item, and the Content Factory verifies the caller's credentials before serving the item, OR

B. Content Factory does not deal with security at all. It is only accessed by trusted components, such as HTMLComposer, that know which items should be visible to which users.

Although both models are adequate in theory, in practice the second one is more error-prone due to the distributed enforcement of access restrictions. It also prevents the Content Factory from publishing some of the items in the file system for fast access by browsers, since the Factory cannot know whether the item is confidential. We have decided some time ago to go with model A.

Assuming that Content Factory is responsible for guarding access to items, there is still an issue with items delivered by URL reference. Delivery by URL reference is convenient when:

- the item is an image, and URL must be imbedded in the page to display it,
- the item is a knowledge base article that you want to show in a separate browser, separate frame, or separate IFrame,
- the item is an attachment and you want the browser to launch a viewer appropriate for the file type.

As implemented in early versions, URL references were not secure. For example, if you guessed the names of other people's email attachments, you could view these attachments by simply pointing your browser to their URLs. This is because browser credentials are verified only when constructing a URL reference, not when serving out data in response to the URL request. A little encryption can make item URLs difficult to guess, but, since URLs are always transmitted in the clear (even when using SSL), it would not stop an adversary who can monitor network traffic.

The HTTP protocol has no built-in means for browser identification. The identification token must be transmitted either as part of posted data, or as a cookie, or as part of the URL. The URL method is not secure against impersonation by a wiretapper, who can record the URL and attach it to another data packet. The cookie method withstands this attack because SSL encrypts all cookies together with the rest of the data packet, so they cannot be separated without decryption. However, the cookie method fails to distinguish a child browser window from its parent, because the two share the same cookie pool. transaction server must distinguish browser windows, however, because they represent separate transaction server sessions.

IIS uses the cookie mechanism to identify the calling browser. Transaction server uses URL identification (so-called "BrowserID") in combination with the IIS feature ("IIS SessionID"). This allows transaction server to securely distinguish browsers running on separate computers, and to insecurely distinguish parent and child browser windows. Consequently, although transaction server may attach different security roles to different browsers, or even to different workflows rendering into the same browser, a little hacking gives you all the permissions of any browser currently running on your computer.

The simplest way to fit secure URL references into this framework is the following. For each secure item, Content Factory produces a URL pointing to ContentGateway.asp and containing the name of the item. The ASP file, when invoked, calls HttpRouter with the request's IIS SessionID, the Router polls all active sessions that match this IIS SessionID, and each HttpSession object attempts to retrieve the item from the Content Factory using the Context of each pending Render as credentials. If any Render within any active session with matching SessionID has access to the Content item, the request is fulfilled. (If a Content item is public, Content Factory can save CPU cycles by constructing static URLs pointing to the file system.)

A more complex approach is to identify the requesting browser and even the requesting workflow instead of stopping at IIS SessionID. This means that the client must modify the URL provided by the Content Factory to insert additional information. The extra complexity, and the issues raised by ASP front ends that do not necessarily maintain BrowserID and DocID, are probably not worth the benefit, especially since, as already discussed, this additional identification is not tamper-proof anyway.

MTS Security

MTS security sits on top of NT security and introduces the concept of Roles. Users and groups can be added to MTS roles. Which roles may invoke which methods on which interfaces can be configured in the MTS management console; this is called "declarative role-based security". Also, an API is available for programs to check at runtime whether a particular user is in a specified MTS role; this is called "programmatic role-based security". Together, these two kinds of role-based security provide an "application level" authentication scheme where applications can be configured in terms of Roles and an administrator can assign specific users and groups to specific Roles.

Programmatic role-based security is especially appropriate for data driven systems, since security may depend on the metaclasses of objects, which are never known until runtime.

SQL Server Security

SQL Server has two types of security - integrated security (i.e. uses the NT login as its login), and the standard login which requires a user ID and password to log in. The latter scheme requires additional administration work and also requires a user to have two different logins - one for NT and one for SQL Server.

Microsoft Decision Support Services (OLAP)

For OLAP, SQL Server uses Roles mapped to the NT security system's objects. Roles are created and managed at the database level. These can then be assigned to specific cubes within the database. You can use the Manage Roles dialog box to assign roles to a specific cube and manage the access privileges for that role as they apply to the cube. Authentication of the user is done based on the logon information of the user. Access control can only be implemented on the NTFS file system. Permissions can be assigned to various

Microsoft Site Server

The Microsoft Site Server provides a service called Membership Services. This is functionality that manages the information on registered users and provides a way to map several users (which could run into the thousands or millions) to physical NT logins (which are limited if we want to keep the domain architecture simple).

Site Server 3.0 stores the user information (Membership Services) in a Directory (LDAP store). This information is not related to NT security in that no NT security accounts are added to the system. Site server can operate in either mode - use native NT security for all user accounts or use its own membership database to authenticate users.

At this time, the complexity of setting up Site Server, with all its planning requirements, seems to big a requirement. We will defer consideration of Site Server to a later date.

Transaction Server Security Architecture

The transaction server's security architecture is illustrated in Fig. 9.

Integrated NT Security

We will use integrated NT security and additional capabilities provided by IIS 910, MTS 920 and SQL Server 930 to implement security in transaction server. All transaction server components will rely on the fact that the user has logged into the system under an existing NT login account and all transactions are performed using those security credentials. This also enables the notion of Single Logon where a user on an Intranet need only log onto their own workstation and be able to access the applications running on transaction server if they are authorized to do so.

The following limitations of Windows NT security restrict what we can do, but also help define the security architecture for transaction server:

- Total number of physical accounts per domain
- Inability to hop machines
- MTS Roles 940

The components in transaction server use MTS Roles 940 to enforce security through transactions. This means programmatically checking the Role of the user to ensure that they are allowed to perform a specific action. The security information for the transaction server will reside in meta-data and can be interpreted by the appropriate components that need to enforce or check the security credentials of the user.

SQL Server

The transaction server will not use DBMS security as a way to restrict access to DBMS objects. Instead, all security validation is done in the business layers so that by the time the call gets to the database, there is no real need to do further validation. This simplifies the administration of the DBMS enormously.

We have two options for SQL Server 930:

- Integrated login - If we use integrated login, this implies that SQL Server 930 has to run on the same machine as the transaction server and IIS 910. However, this is not necessarily the recommended approach if we want the system to scale - typically, SQL Server 930 must run on its own machine that is configured and tuned for SQL Server 930.
- Basic login - In this case, we have to provide logins for transaction server to use when data it accessed from SQL Server 930.

It makes most sense to use basic login for SQL Server 930and any other DBMS because we want to have a dedicated machine for the DBMS and also because it is not necessarily true that all DBMSs support integrated NT security (Oracle does support NT integrated security). Also, when we do integrations with mainframe systems, additional logins will be required anyway.

This means that there will be a place where userid and password information is kept. The DB component in transaction server, which is responsible for the DBMS login, will pull the information and use that login information to get to the DBMS. The DBMS will be configured to have accounts for those users.

In order to take advantage of connection pooling, single <user><password> combinations will be used for many/all the users. This is a reasonable approach in a server environment where the users have been validated by layers higher in the calling hierarchy.

Microsoft Decision Support Services (OLAP)

Since OLAP uses integrated NT security, we can just use the impersonation that IIS 910 performs and that MTS 920 propagates. On the administration side, Roles have to be set up in SQL Server 930 to control access to the various cubes that have been created in the database.

IIS and Internet Explorer

If we are using Site Server Membership Services, we will need to take the user's login information and get it converted to NT security credentials for use by transaction server. This is most probably built into Site Server and specific implementation details need to be determined. Once the login is authenticated, transaction server can proceed to use strictly MTS Roles 920 to validate the actions of the user.

Details

On the web channel, we will prompt the user for login information, using the challenge-response mechanism. IE 950 will not send the plain text password across the network; Netscape 960 will. For the Netscape case, SSL can be employed to protect the clear-text passwords that go across the Internet.

We need to ensure that a mechanism is available for users using other channels, e.g. email, CTI, etc. to log into the application before they try to perform any transactions.

IIS 910 translates a user login (using username and password) into an application login (using NT security). From that point on, the user session has NT security credentials associated with it and the workflow context will carry the user's ID.

Specific security information for the application is placed in the meta-data.

There is user-specific information in the database that pertains to queues, preferences, etc. The userID used in this database must match the userID that is used in the NT security accounts - or, there has to be a mapping from one to the other.

SQL Server 930 and other DBMSs are configured with logins that are only used by transaction server components. No user goes directly to the DBMS. Therefore, there could be one system-wide login into the DBMS that all transaction server components can use - this is the preferred mode so that DBMS connections can be pooled.

Features Supported

Secure transaction server system that can validate user access to functions, data, etc.

Users can change their password - This capability will not be available in the first release.

Support thousands of users - NT4 has limits on accounts per domain, which will make scaling difficult to administer. However, Windows 2000 will alleviate this problem and let us scale much higher with a single domain.

Administration required: set up NT accounts for users and groups; set up user IDs in the database to match the NT users; set up login(s) to the DBMS for use by transaction server components. The hardest part is keeping the NT security information in sync with the user IDs in the database. We will provide a tool to do this for new accounts.

The administration information needs to be documented really well even if all it does is refer to the Microsoft manuals.

How The Pieces Fit Together

The application defines what Roles are needed for the correct operation and access control of the parts of the application.

User and Group accounts are present in the NT security system. These are done through the standard NT User Manager for Domains.

Corresponding userIDs need to be put into the database.

OLAP Roles 970 are created and users/groups assigned to them.

DBMS logins are created and the information is placed in the Directory in a well-known and secure place.

When a user connects to transaction server and needs to perform an operation that requires a login, transaction server prompts the Browser to get this information and then IIS 910 logs in the user.

Deployment Architecture

All processes on Windows NT run under specific security accounts. It is possible to configure Windows NT services and COM components to run as specific users. Other processes typically run as the user that is logged in.

There are specific requirements for configuring transaction server components due to the fact that the transaction server is tightly integrated with Windows NT security. If the system is not set up correctly, failures can occur due to access violations.

The diagram of Fig. 10 displays the physical architecture of a typical transaction server system that supports email processing.

"Account" object 1010 indicates what processes need to be configured in specific accounts. MTS process 1020 on the transaction server machine and the Agent process 1030 on the Exchange machine must all run in the same Windows NT account. This is because they communicate with each other through the queue (MSMQ), which is a private queue.

The Exchange service must run as a user that has Domain Administration privileges - this is a requirement for Microsoft Exchange.

Security Considerations

The transaction server also requires privileges that allow it to create, inspect and destroy login accounts for users. It also must be able to examine all the inboxes that reside on the Exchange server, and so require domain-level privileges, rather than machine-level privileges. Communication between the Exchange Agent and the transaction server is done via a private message queue, which requires that the same user accesses both ends of the queue.

It is highly recommended that all the machines that run transaction server services live in their own domain. This includes the Exchange server, the IIS server(s), the transaction server server(s), and any other servers (file, domain controllers, etc.) that are used in that environment. The DBMS server(s) can be in a different domain, if that is necessary. This is especially true if Internet access is provided to the transaction server system. On an intranet, this may not be as important.

Corporations typically have either a single-domain architecture, if they have few users, or a multi-domain architecture where the security domains typically trust each other. There may be other domains that only trust the master domain(s) but which are not themselves trusted. A transaction server domain could be configured in this way, if trusting it is an issue.

This gives us the security configuration displayed in Fig. 11.

The following accounts then need to be set up in the "transaction server Domain" (it can be named whatever you wish):

A Domain Admin account for the Exchange Server.

A Domain Admin account for the MTS and Logger processes on the transaction server server(s), and the Exchange Agent service on the Exchange server.

This transaction server account must be given access to all InBoxes on the Exchange server. This is done through the Exchange administration program.

IIS configures itself to run under a specific account - it is not necessary to change this setting.

Of course, on an Intranet, this may not be necessary as all the users are authorized to be on the network and can belong to the same domain.

Additional Levels Of Security

A customer could provide stricter security using the following mechanisms:

A secure web site - SSL can be used to force every user to log in before they can access the web site. Then, all data transferred between the browser and IIS would be encrypted.

Instead of a logon screen, customers could require that their users have things like Certificates or Smart Cards to identify themselves. IIS and, by association, transaction server supports certificates as a way to identify the user. This is all transparent to the transaction server as it is taken care of by "the system".

Object-level security at the DBMS level - We could design our login information in such a way that every user or group would have their own login into the database. A customer could then use DBMS security to restrict access to the DBMS' objects. This is not recommended due to the scalability impact of not using pooled DBMS connections.

Some embodiments use a different model of security that takes advantage of Microsoft Site Server Membership services. This allows us to separate the management of user accounts from the security system, thus allowing us to support many millions of users without the administrative nightmare of maintaining Windows NT domain accounts.

The present disclosure is merely illustrative in nature. Embodiments other than those described herein are possible, according to the principles of the present invention, as described in the following claims.

CLAIMS

We claim:

1. A computer system for modeling business workflows, the computer system comprising:

a server computer;

one or more client computers connected to the server computer via a computer network; and

an application program executed by the server computer, wherein the application program further comprises:

a plurality of business workflows to control the application program using business rules to coordinate and sequence operations;

a plurality of business information objects to store data used by the application program;

a plurality of mappers to transfer data between a persistent storage and the business information objects;

a plurality of interactors to render workflows and business information objects on external delivery channels;

a synchronizer to control and sequence operations of the application program; and

a peripheral controller to orchestrate the cooperation of the application program with external devices and agents.

2. The computer system of claim 1, wherein the plurality of business information objects further comprise a plurality of persistent business information objects stored in a storage repository.

3. The computer system of claim 1, wherein the mappers perform data transformations.

4. The computer system of claim 3, wherein the mappers transfer information between databases and business information objects.

5. The computer system of claim 3, wherein the mappers use data organization trees to transfer information to and from the business information objects.

6. The computer system of claim 1, wherein the delivery channels further comprise the Web, e-mail services and telephony.

7. The computer system of claim 1, wherein the synchronizer is an event machine.

8. The computer system of claim 1, wherein the external devices and agents further comprise CTI and e-mail services.

9. A method of modeling business workflows, the method comprising:

a plurality of business workflows controlling an application program using business rules to coordinate and sequence operations;

a plurality of business information objects storing data used by the application program;

a plurality of mappers transferring data between a persistent storage and the business information objects;

a plurality of interactors rendering workflows and business information objects on external delivery channels;

a synchronizer controlling and sequencing operations of the application program; and a peripheral controller orchestrating the cooperation of the application program with external devices and agents.

10. The computer system of claim 9, wherein the plurality of business information objects further comprise a plurality of persistent business information objects stored in a storage repository.

11. The computer system of claim 9, wherein the mappers perform data transformations.

12. The computer system of claim 11, wherein the mappers transfer information between databases and business information objects.

13. The computer system of claim 11, wherein the mappers use data organization trees to transfer information to and from the business information objects.

14. The computer system of claim 9, wherein the delivery channels further comprise the Web, e-mail services and telephony.

15. The computer system of claim 9, wherein the synchronizer is an event machine.

16. The computer system of claim 9, wherein the external devices and agents further comprise CTI and e-mail services.

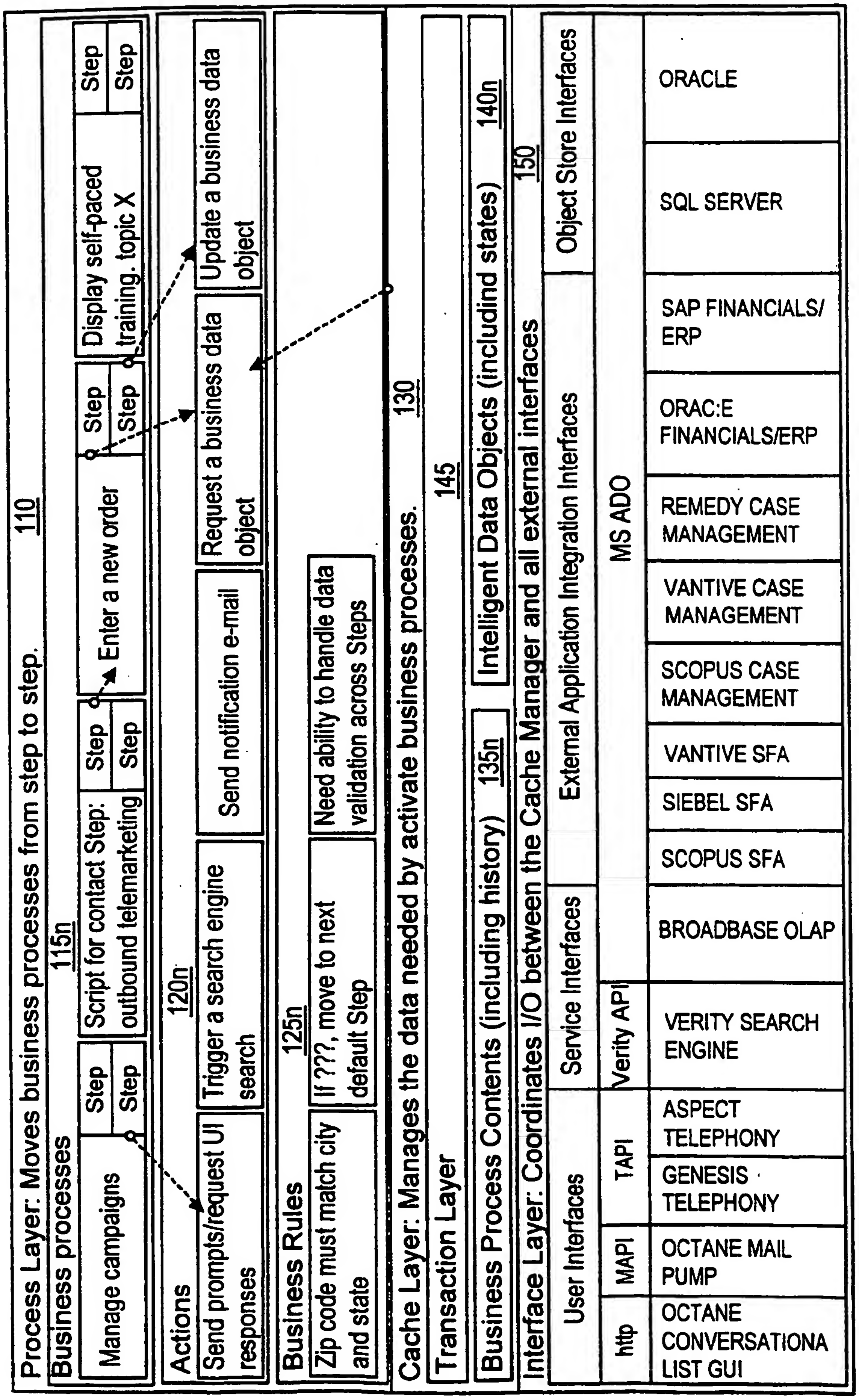

Process Layer: Moves business processes from step to step. 110
Business processes 115n
Manage campaigns
Step
Step
Script for contact Step: outbound telemarketing
Step
Step
Enter a new order
Step
Step
Display self-paced training. topic X
Step
Step
Actions 120n
Send prompts/request UI responses
Trigger a search engine search
Send notification e-mail
Request a business data object
Update a business data object
Business Rules 125n
Zip code must match city and state
If ???, move to next default Step
Need ability to handle data validation across Steps
Cache Layer: Manages the data needed by activate business processes. 130
Transaction Layer 145
Business Process Contents (including history) 135n
Intelligent Data Objects (includind states) 140n
Interface Layer: Coordinates I/O between the Cache Manager and all external interfaces 150
User Interfaces
Service Interfaces
External Application Integration Interfaces
Object Store Interfaces
http
MAPI
TAPI
Verity API
MS ADO
OCTANE CONVERSATIONA LIST GUI
OCTANE MAIL PUMP
GENESIS TELEPHONY
ASPECT TELEPHONY
VERITY SEARCH ENGINE
BROADBASE OLAP
SCOPUS SFA
SIEBEL SFA
VANTIVE SFA
SCOPUS CASE MANAGEMENT
VANTIVE CASE MANAGEMENT
REMEDY CASE MANAGEMENT
ORAC:E FINANCIALS/ERP
SAP FINANCIALS/ ERP
SQL SERVER
ORACLE

Fig. 1

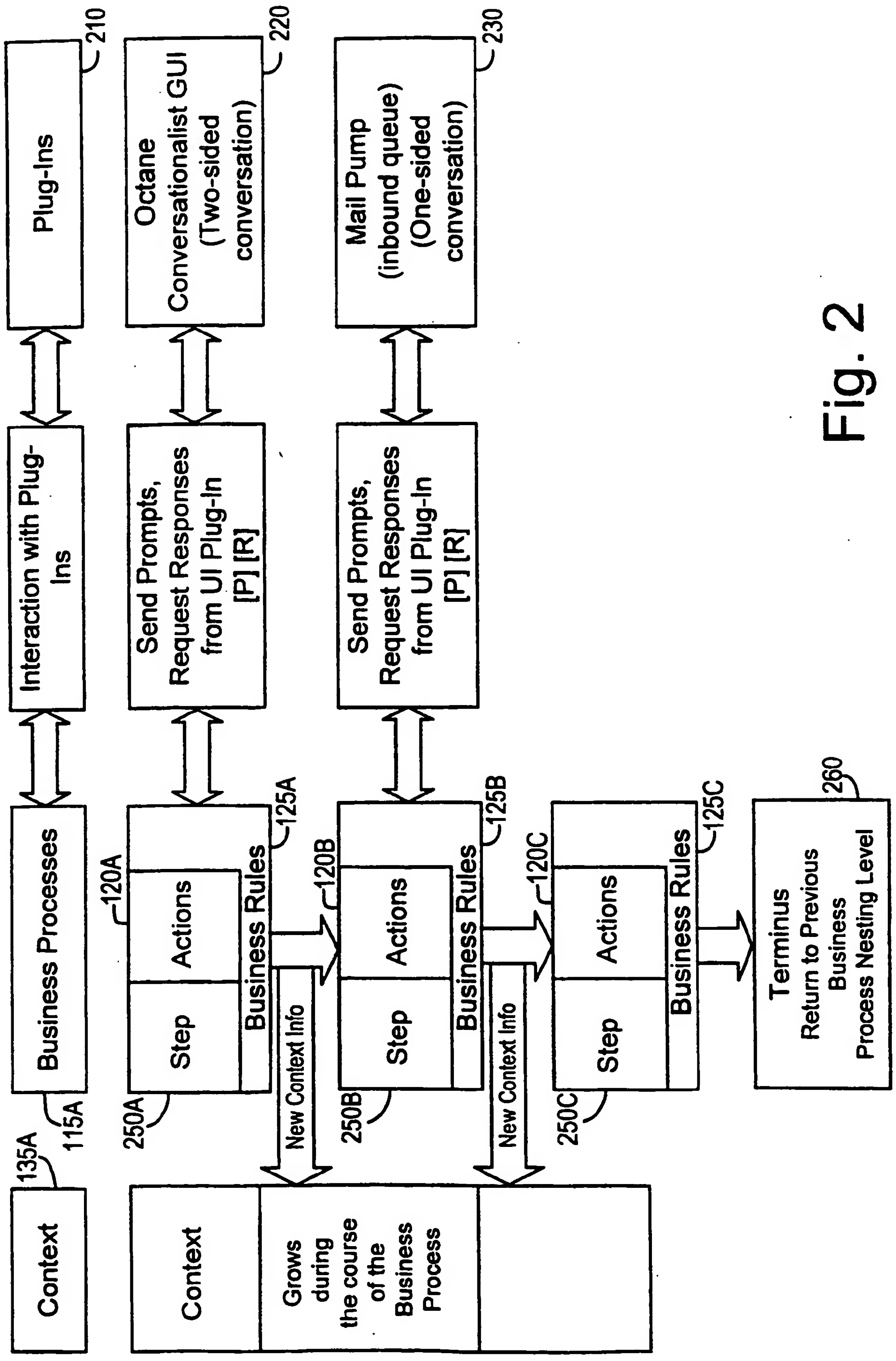

Plug-Ins
210
Octane Conversationalist GUI (Two-sided conversation)
220
Mail Pump (inbound queue) (One-sided conversation)
230
Interaction with Plug-Ins
Send Prompts, Request Responses from UI Plug-In [P] [R]
Send Prompts, Request Responses from UI Plug-In [P] [R]
Business Processes
115A
Step
Actions
Business Rules
120A
125A
250A
Step
Actions
Business Rules
120B
125B
250B
Step
Actions
Business Rules
120C
125C
250C
Terminus Return to Previous Business Process Nesting Level
260
New Context Info
New Context Info
Context
135A
Context
Grows during the course of the Business Process

Fig. 2

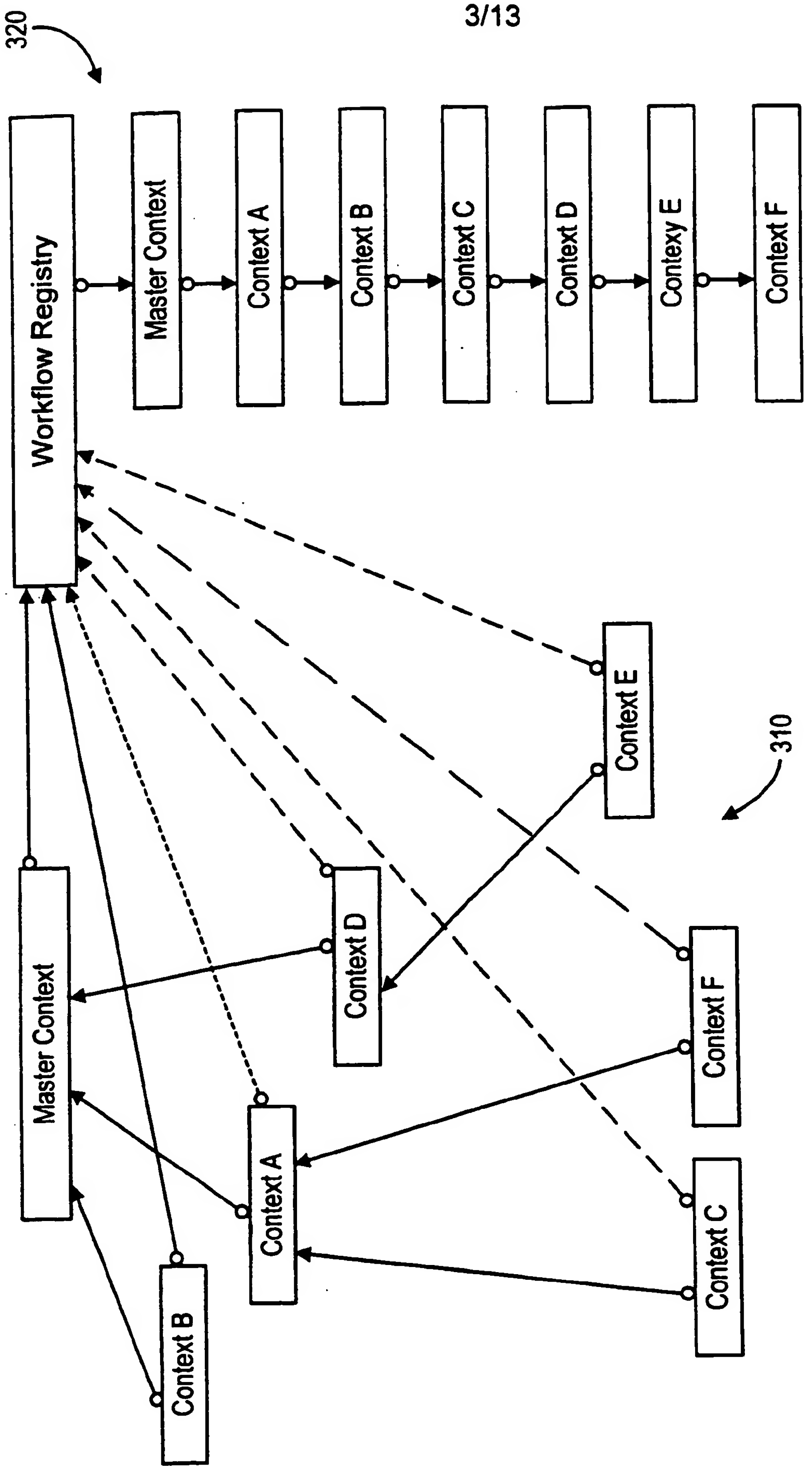

320
Workflow Registry
Master Context
Context A
Context B
Context C
Context D
Contexy E
Context F
Master Context
Context D
Context E
Context A
Context F
Context B
Context C
310

Fig. 3

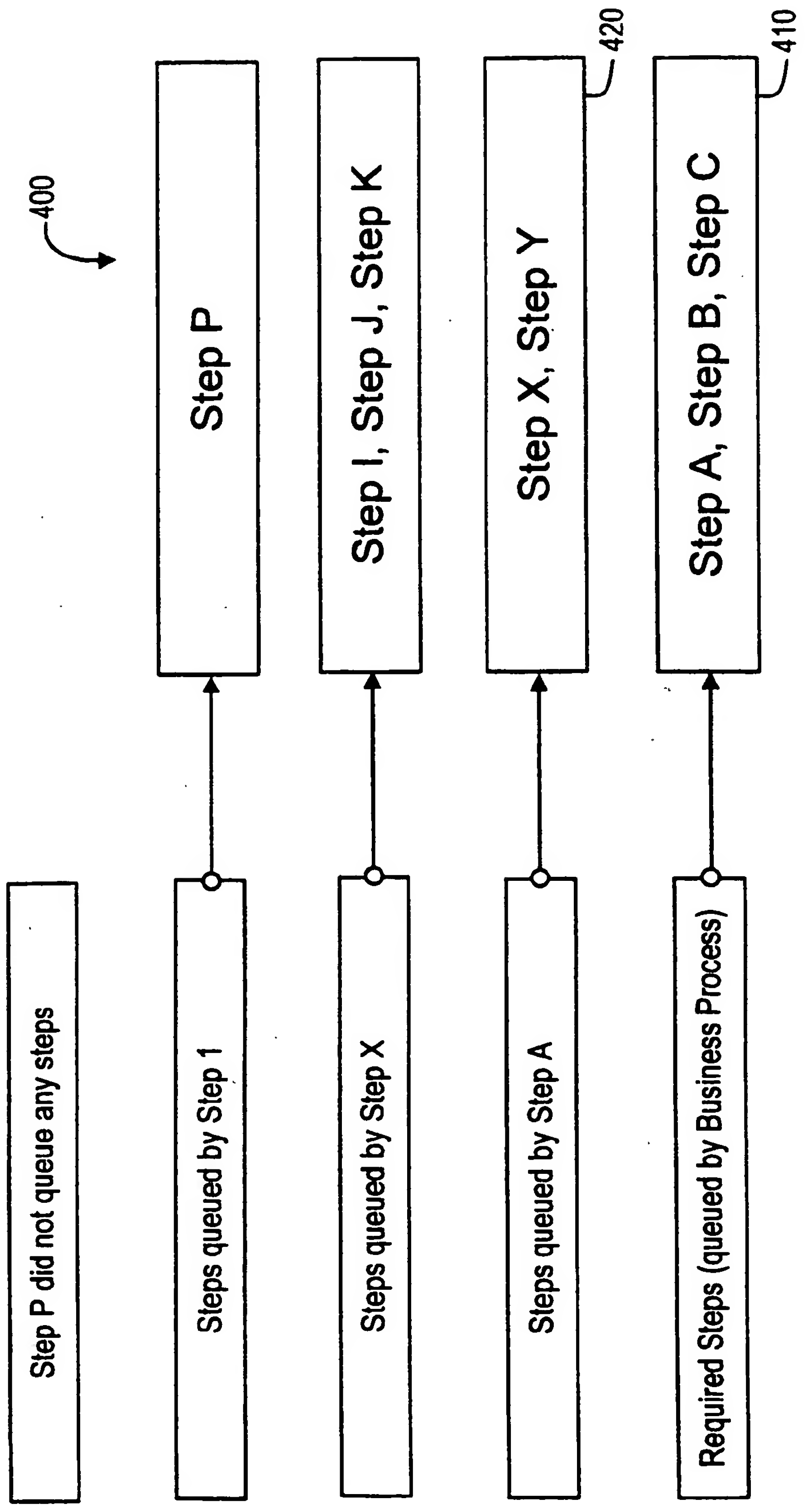

410
Required Steps (queued by Business Process)
Step A, Step B, Step C
420
Steps queued by Step A
Step X, Step Y
Steps queued by Step X
Step I, Step J, Step K
Steps queued by Step 1
Step P
Step P did not queue any steps
400

Fig. 4

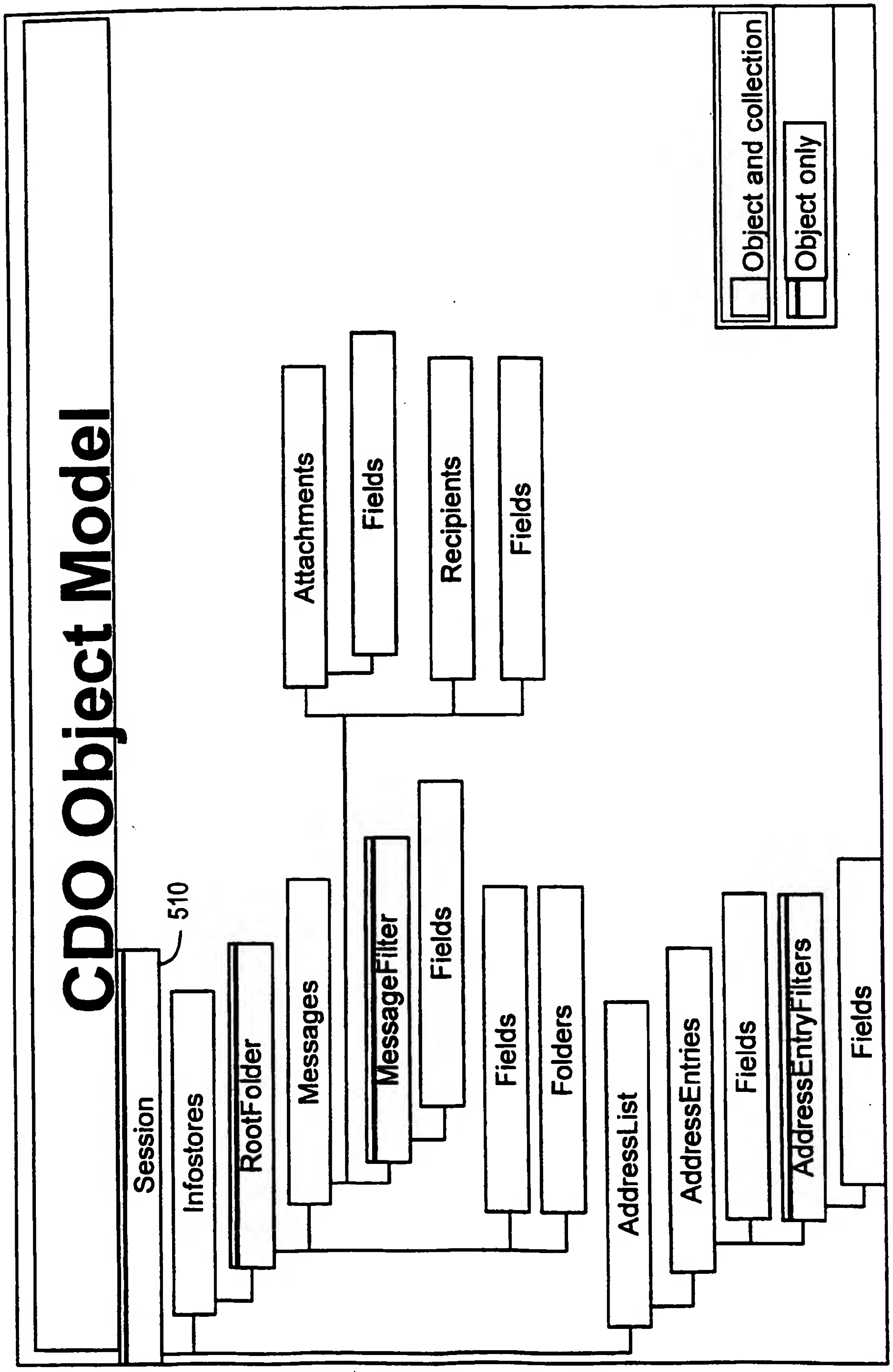

CDO Object Model
510
Session
Infostores
RootFolder
Messages
MessageFilter
Fields
Attachments
Fields
Recipients
Fields
Fields
Folders
AddressList
AddressEntries
Fields
AddressEntryFilters
Fields
Object and collection
Object only

Fig. 5

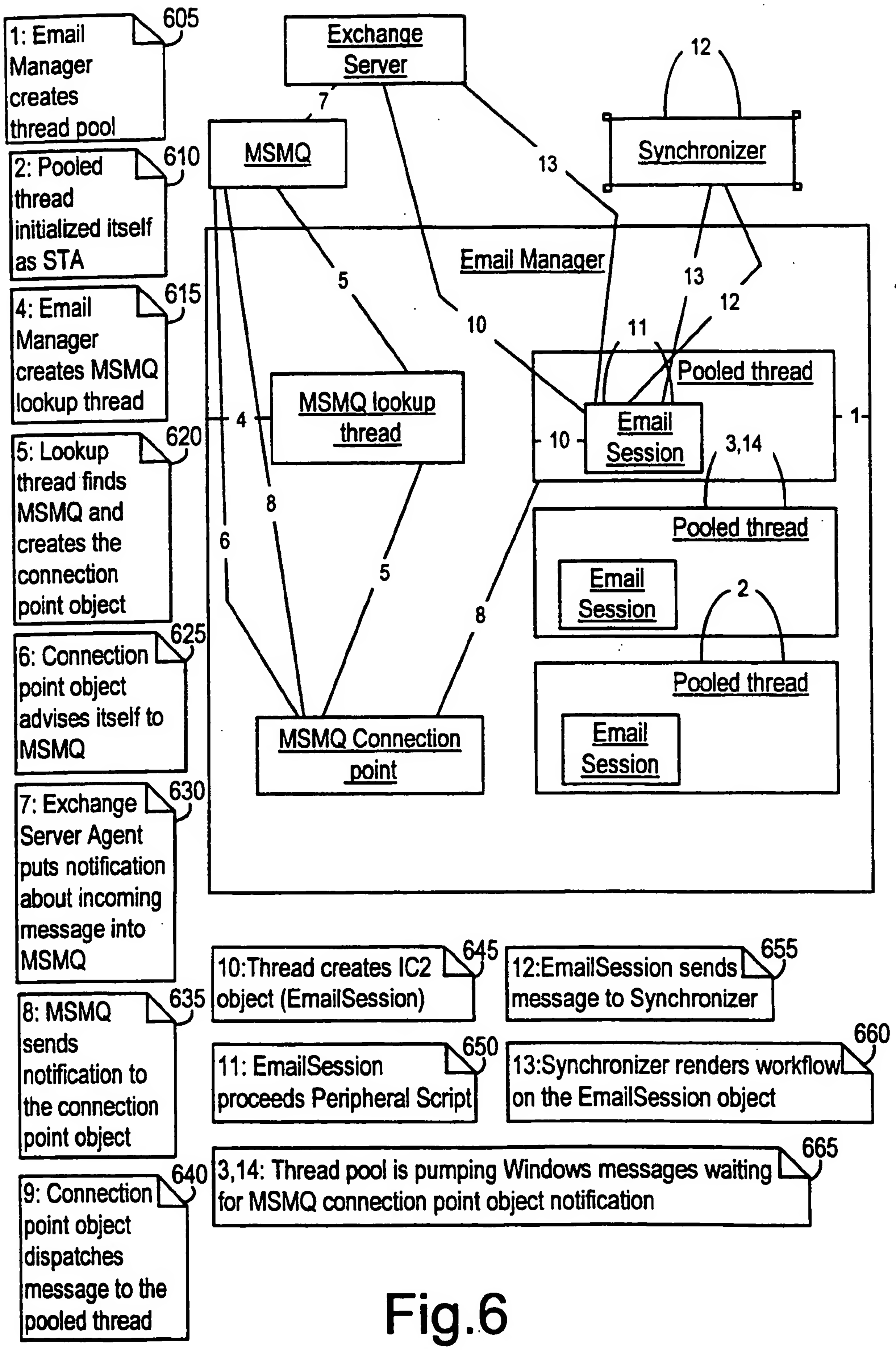

1: Email Manager creates thread pool
605
2: Pooled thread initialized itself as STA
610
4: Email Manager creates MSMQ lookup thread
615
5: Lookup thread finds MSMQ and creates the connection point object
620
6: Connection point object advises itself to MSMQ
625
7: Exchange Server Agent puts notification about incoming message into MSMQ
630
8: MSMQ sends notification to the connection point object
635
9: Connection point object dispatches message to the pooled thread
640
Exchange Server
MSMQ
Synchronizer
Email Manager
MSMQ lookup thread
MSMQ Connection point
Pooled thread
Email Session
Pooled thread
Email Session
Pooled thread
Email Session
1
2
3,14
4
5
6
7
8
10
11
12
13
10:Thread creates IC2 object (EmailSession)
645
11: EmailSession proceeds Peripheral Script
650
12:EmailSession sends message to Synchronizer
655
13:Synchronizer renders workflow on the EmailSession object
660
3,14: Thread pool is pumping Windows messages waiting for MSMQ connection point object notification
665

Fig.6

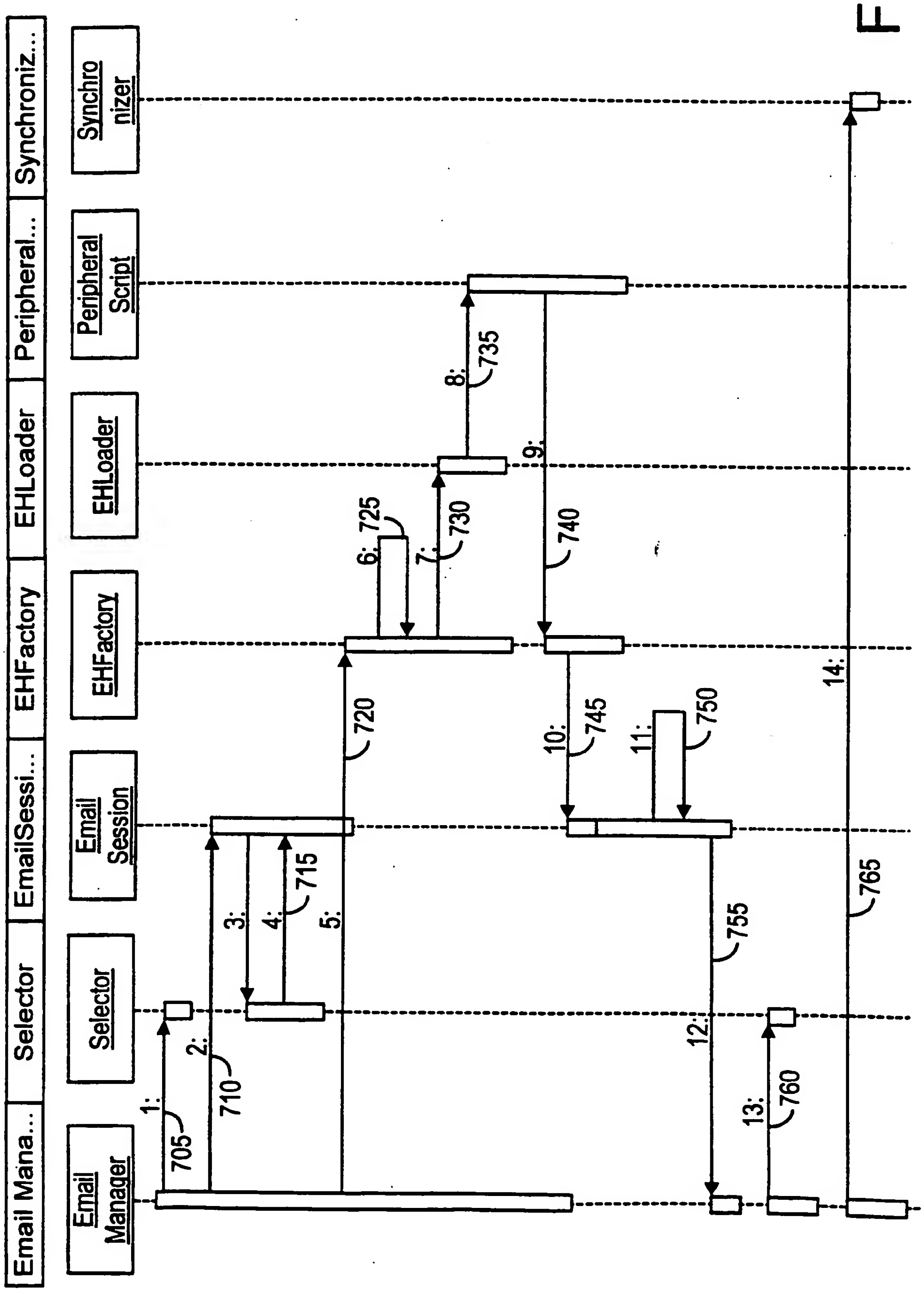

Email Mana...
Selector
EmailSessi...
EHFactory
EHLoader
Peripheral...
Synchroniz...
Email Manager
Selector
Email Session
EHFactory
EHLoader
Peripheral Script
Synchro nizer
1:
705
2:
710
3:
4:
715
5:
720
6:
725
7:
730
8:
735
9:
740
10:
745
11:
750
12:
755
13:
760
14:
765

Fig.7

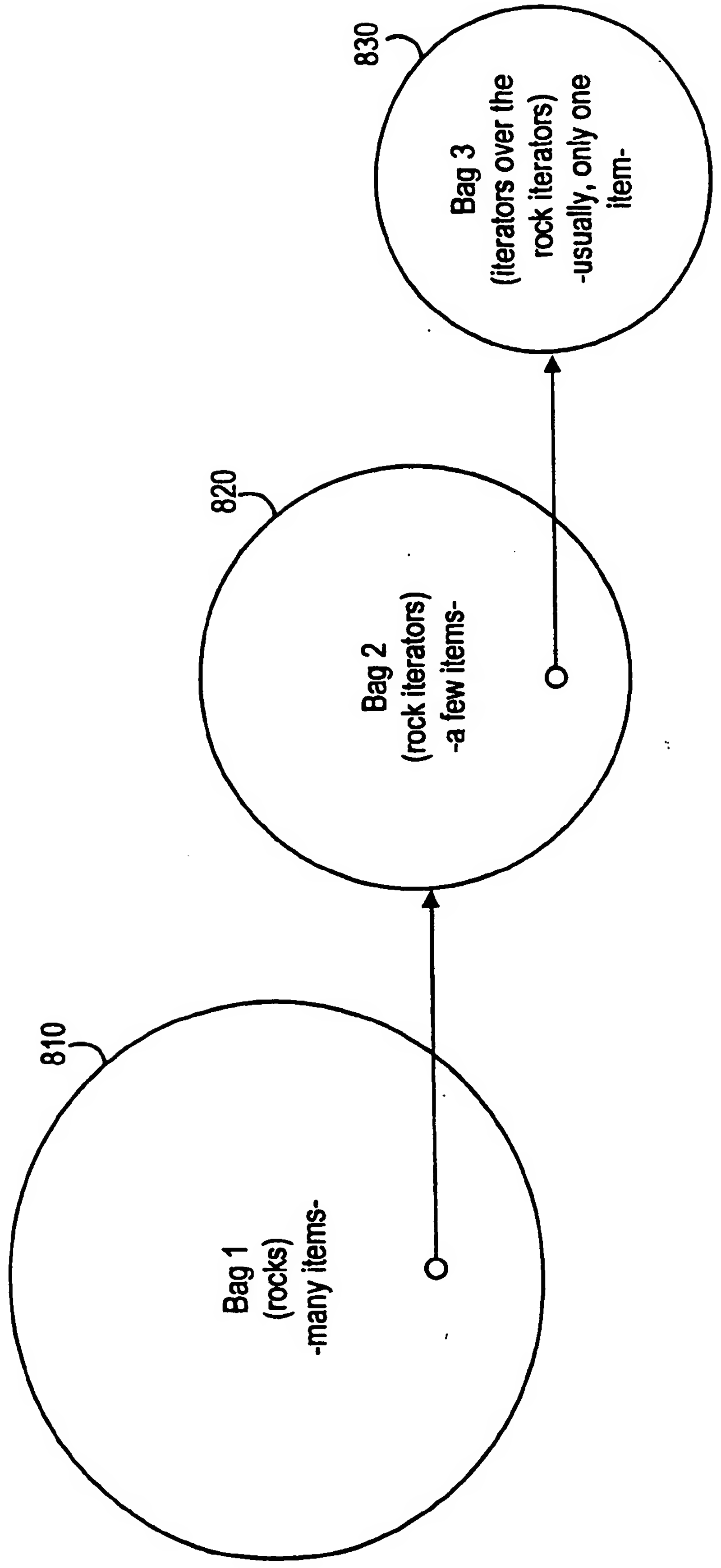

830
Bag 3
(iterators over the
rock iterators)
-usually, only one
item-
820
Bag 2
(rock iterators)
-a few items-
810
Bag 1
(rocks)
-many items-

Fig. 8

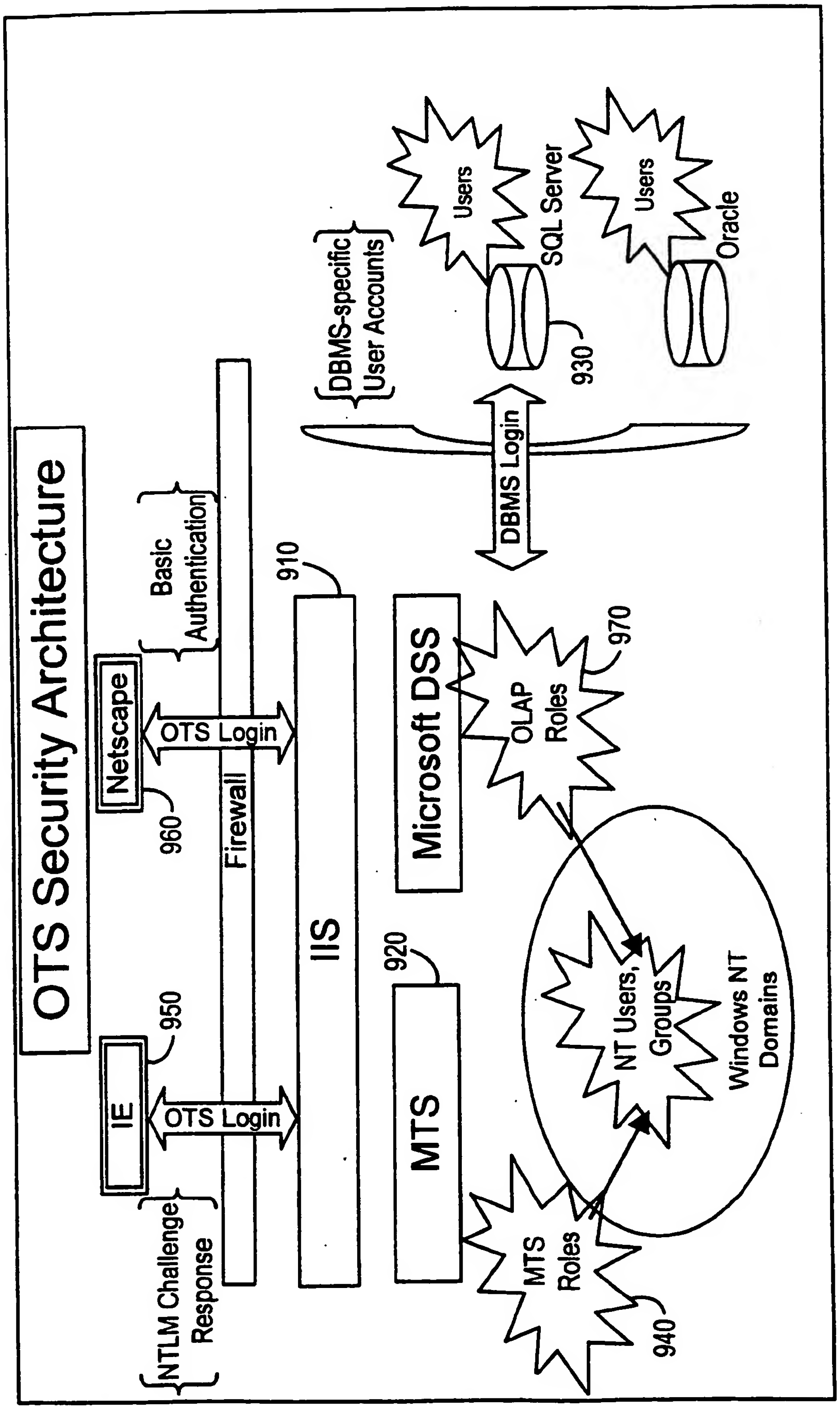

OTS Security Architecture
NTLM Challenge Response
IE
950
OTS Login
Netscape
960
OTS Login
Basic Authentication
Firewall
IIS
910
MTS
920
Microsoft DSS
DBMS Login
DBMS-specific User Accounts
930
Users
SQL Server
Users
Oracle
MTS Roles
940
OLAP Roles
970
NT Users, Groups
Windows NT Domains

Fig. 9

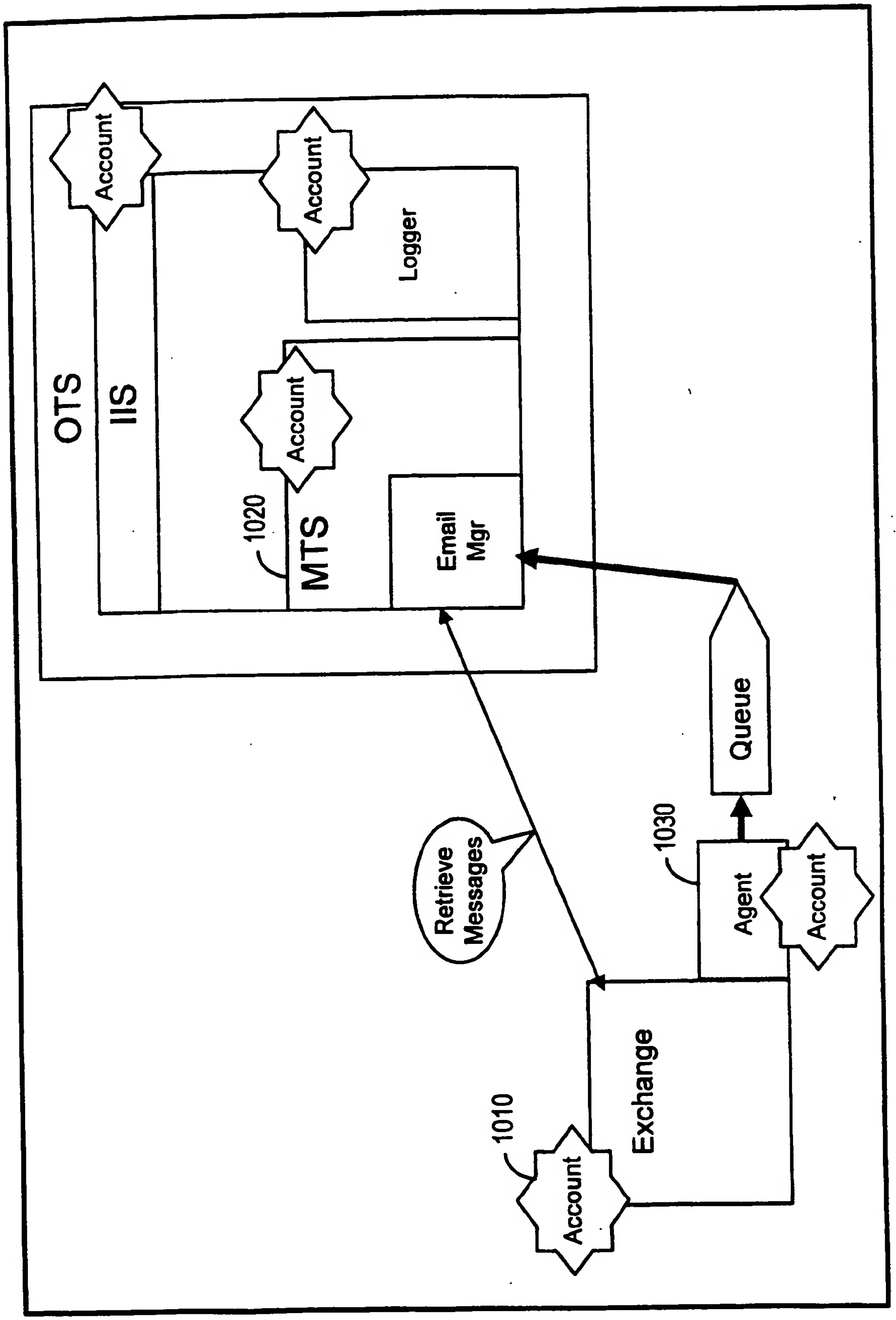

OTS
IIS
Account
Account
Logger
Account
1020
MTS
Email Mgr
Queue
Retrieve Messages
1030
Agent
Account
1010
Account
Exchange

Fig. 10

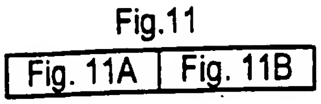


Fig.11
Fig. 11A
Fig. 11B

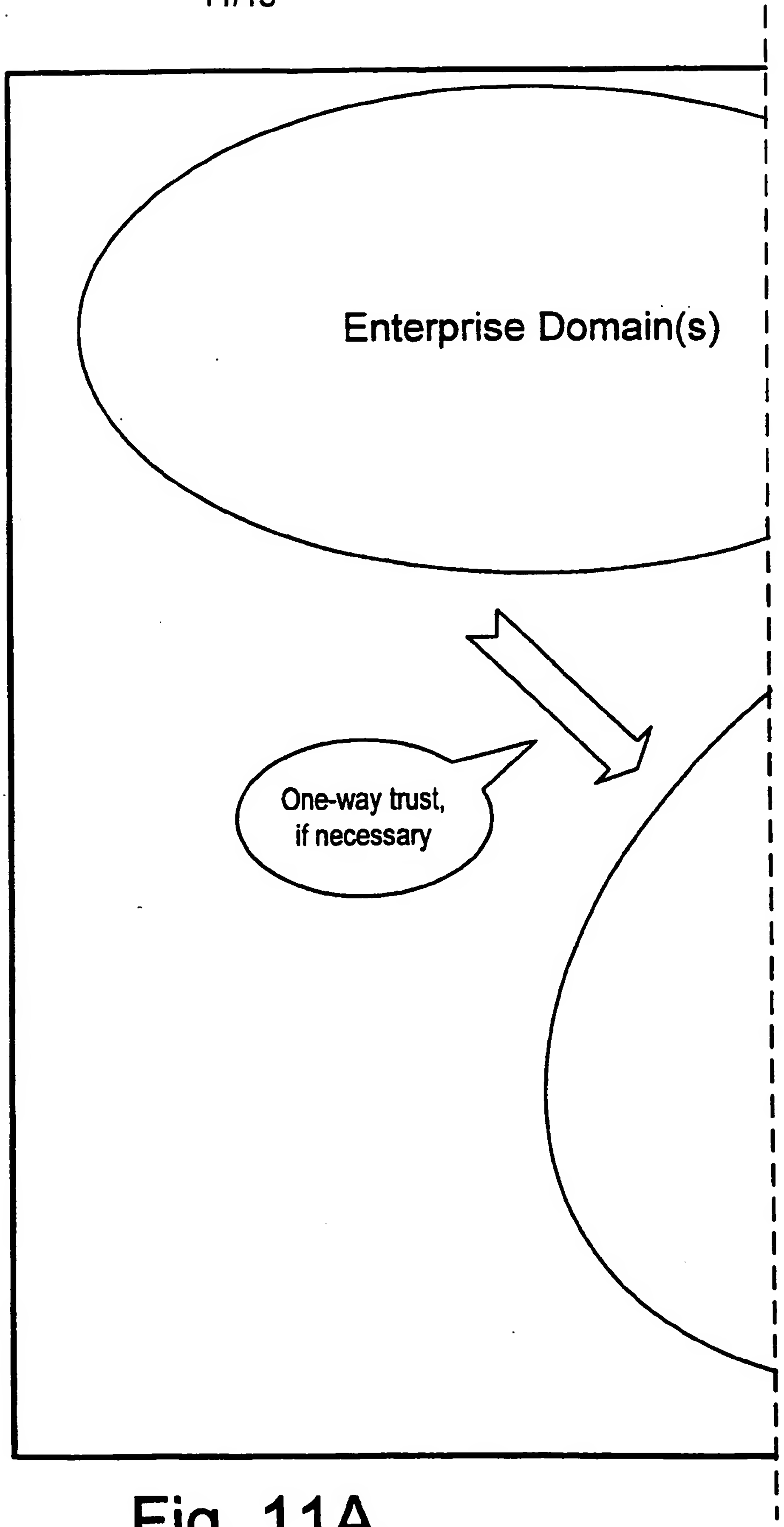

Enterprise Domain(s)
One-way trust, if necessary

Fig. 11A

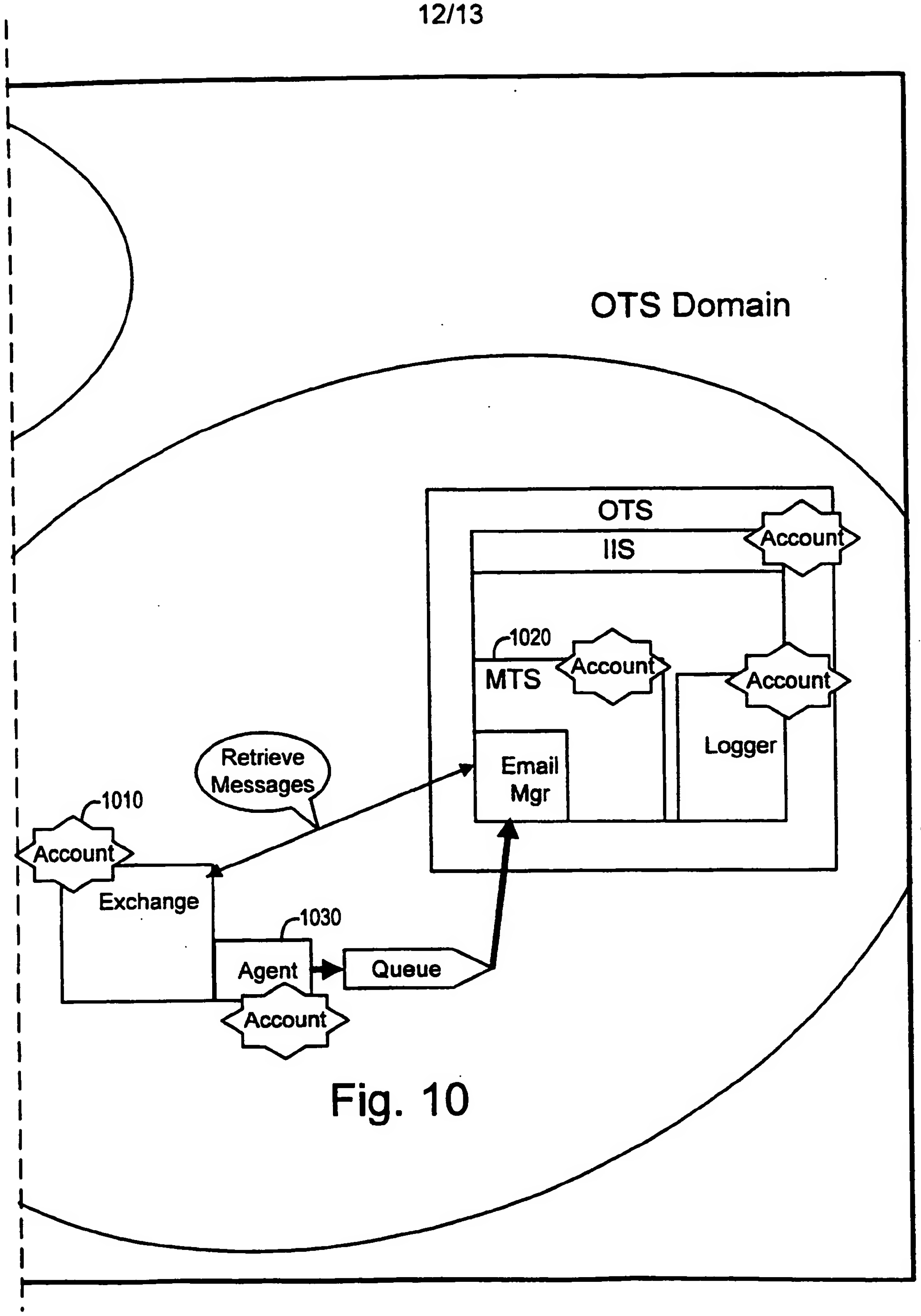


Fig. 10

Fig. 11B

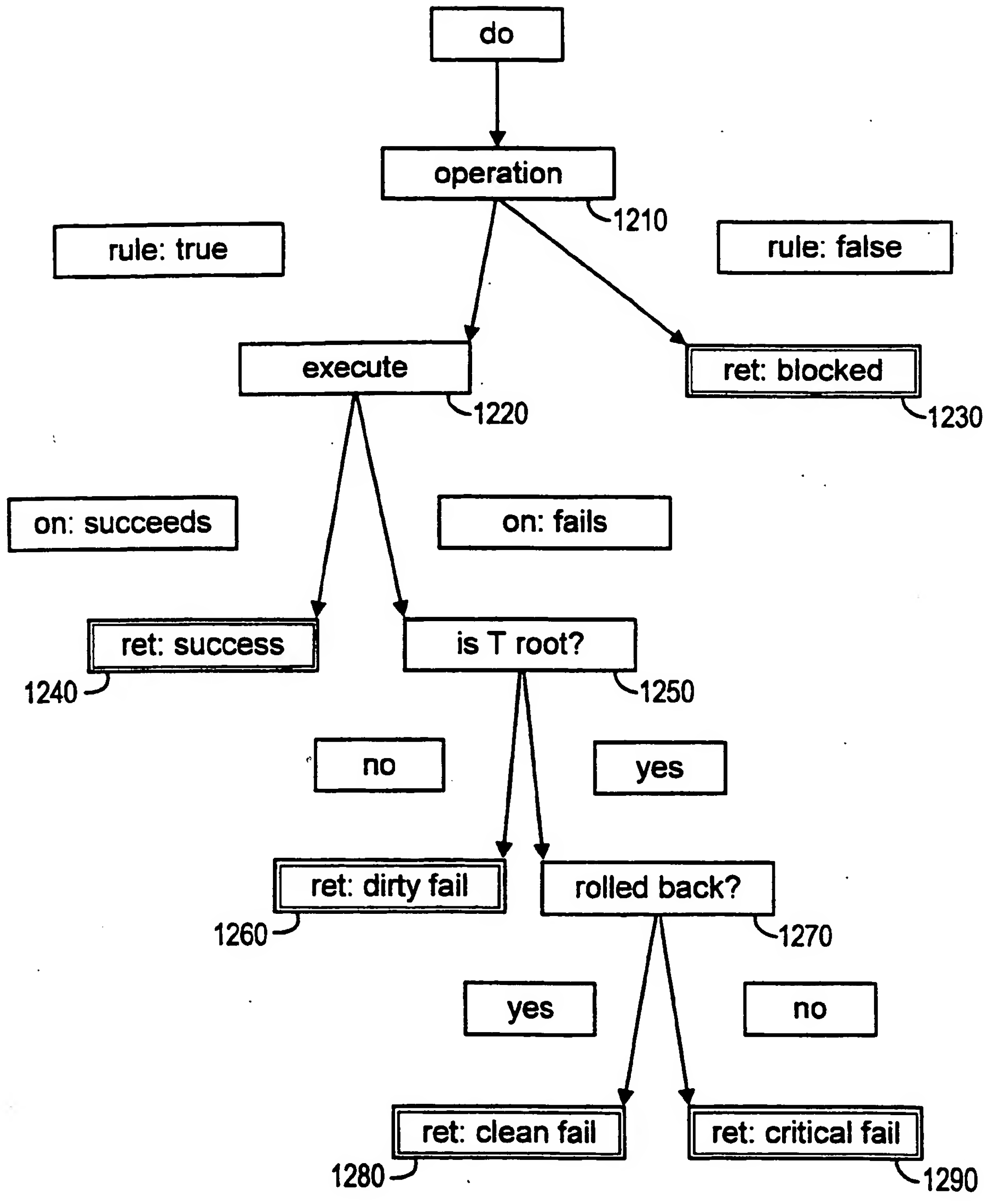

do
operation
1210
rule: true
rule: false
execute
1220
ret: blocked
1230
on: succeeds
on: fails
ret: success
1240
is T root?
1250
no
yes
ret: dirty fail
1260
rolled back?
1270
yes
no
ret: clean fail
1280
ret: critical fail
1290

Fig.12

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/07795

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) :G06F 17/50, 7/60
US CL :703/1, 2, 6; 705/7, 8, 9, 10; 707/100
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED
Minimum documentation searched (classification system followed by classification symbols)
U.S. : 703/1, 2, 6; 705/7, 8, 9, 10; 707/100

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EAST, WEST, IEEE, DIALOG
search terms: business, model, object, event

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5,734,837 A (FLORES et al) 31 March 1998, ALL	1-16
A	MENDELSON. Diagramming Tools. PC Magazine. February 1996.	1-16
A	TAN et al. Improving the Reusability of Program Specification through Data Flow Modeling. IEEE Fifth International Conference on Computing and Information. May 1993. Pages 479-483.	1-16
A	LANG et al. Modeling Business Rules with Situation/Activation Diagrams. IEEE Proceedings of the 13th International Conference on Data Engineering. April 1997. Pages 455-464.	1-16

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

* Special categories of cited documents:	
"A" document defining the general state of the art which is not considered to be of particular relevance	"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"E" earlier document published on or after the international filing date	"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"O" document referring to an oral disclosure, use, exhibition or other means	
"P" document published prior to the international filing date but later than the priority date claimed	"&" document member of the same patent family

Date of the actual completion of the international search 30 JUNE 2000	Date of mailing of the international search report 25 JUL 2000
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer KEVIN TESKA For [illegible] Telephone No. (703) 305-9704

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/07795

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	GRUHN et al. Workflow Management Based on Process Model Repositories. IEEE Proceedings of the 1998 International Conference on Software Engineering. April 1998. Pages 379-388.	1-16
A	RIEMER. A Process-Driven, Event-Based Business Object Model. IEEE Proceedings of the Second International Enterprise Distributed Computing Workshop. November 1998. Pages 68-74.	1-16
A	LOOS et al. Object-Orientation in Business Process Modeling through Applying Event Driven Process Chains (EPC) in UML. IEEE Proceedings of the Second International Enterprise Distributed Object Computing Workshop, 1998. November 1998. Pages 102-112.	1-16